

For Reference

NOT TO BE TAKEN FROM THIS ROOM

For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex LIBRIS
UNIVERSITATIS
ALBERTAENSIS





Digitized by the Internet Archive
in 2019 with funding from
University of Alberta Libraries

<https://archive.org/details/Rolfson1965>

THESES
1965-1966
103

THE UNIVERSITY OF ALBERTA

A COMPILING TECHNIQUE FOR THE IVERSON LANGUAGE

by

Cordell Rolfson

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SEPTEMBER, 1965

UNIVERSITY OF ALBERTA
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read,
and recommend to the Faculty of Graduate Studies for
acceptance, a thesis entitled A COMPILING TECHNIQUE
FOR THE IVERSON LANGUAGE submitted by Cordell Rolfson
in partial fulfilment of the requirements for the degree
of Master of Science.

ABSTRACT

This thesis is concerned with the implementation of Iverson's language via a compiler.

The basic concepts of the language are reviewed together with its application to the logical description of digital machines. Conventional compiling techniques, both direct methods and syntactically oriented methods, are discussed. Some problems encountered in applying these to the scan of Iverson's language are enumerated. Modifications to a direct scan method, which allow compilation of the language are introduced.

The requirements imposed by Iverson's language upon the object time organization of the computer are presented and various methods of data representation, are discussed. A modification to these methods, which allow fulfilment of the above requirements is introduced.

A description of a "pilot model" compiler which uses the above techniques is contained in the Appendix.

ACKNOWLEDGEMENTS

I would like to extend my sincere thanks to Professor W.S. Adams who has provided invaluable guidance to me in both the research incidental to this thesis and in the preparation of it.

I would also like to thank Dr. D.B. Scott for the environment in which I have been able to work and to Dr. J.J. McNamee for his encouragement.

TABLE OF CONTENTS

	Page
CHAPTER I	Introduction
Section 1.1	Algorithmic Languages
Section 1.2	Iverson's Language
Section 1.3	The Iverson Notation in Logic Design
Section 1.4	Machine Simulation at a Logical Structure Level
Section 1.5	Automatic Execution of Iverson's Language
CHAPTER II	Conventional Compiling Techniques
Section 2.1	Vehicles for the Description of Algorithms
Section 2.2	Compilers and Compiling Techniques
Section 2.3	Direct Methods
Section 2.3.1	Stacks
Section 2.3.2	The Stack in a Compiler
Section 2.3.3	Evaluation of a Lukasiewicz String
Section 2.4	Syntactically Oriented Techniques
Section 2.4.1	Syntactic Routine Methods
Section 2.4.2	Syntactic Information Methods
CHAPTER III	Scan of an Iverson String
Section 3.1	Problems in the Scan of Iverson's Language
Section 3.1.1	Transliteration
Section 3.1.2	Structural Parameters
Section 3.1.3	Special Operands
Section 3.1.4	Operators

Section 3.1.5	Indices	34
Section 3.1.6	Declarations	34
Section 3.2	Direct Scan of Iverson's Language	35
Section 3.2.1	The Use of Syntactically Oriented Techniques on Iverson's Language	39
Section 3.3	Generators	40
CHAPTER IV	Machine Organization at Object Time	42
Section 4.1	Dynamic Storage Allocation	42
Section 4.2	Organization for Selection Operations	44
Section 4.3	Representation of Data	44
Section 4.3.1	Linear Representations	45
Section 4.3.2	The Explicit Grid Matrix	46
Section 4.3.3	Chained Representation	47
Section 4.4	Pools	48
Section 4.5	Representation of the Structured Operands of Iverson's Notation	49
CHAPTER V	Concluding Remarks	52
Section 5.1	Review of the Thesis	52
Section 5.2	Future Investigations	53
BIBLIOGRAPHY		54
APPENDIX A		60
APPENDIX B		62
APPENDIX C		64

APPENDIX D	69
APPENDIX E	71
APPENDIX F	73

LIST OF TABLES

TABLE 3.1	Values assigned to precedence vector	38
TABLE B.1		63
TABLE F.1		74
TABLE F.2		75

CHAPTER I

Introduction

Section 1.1 Algorithmic Languages

The electronic digital computer has become a tool of major importance to many scientific disciplines (if only as a status symbol). The efficient use of machines in any particular discipline has been conditioned by the existence of language problems at two levels.

- 1) The numerical languages of computer are poorly adapted to human communication

and

- 2) several different mathematical notations may be in use simultaneously in a given field.

In theory, both numerical computer languages and the mathematical notations are supposed to describe the same (numerical) processes which have been found useful to a given discipline. In practice, however, translating from existing mathematical algorithms to equivalent algorithms acceptable to machines has proved to be difficult and often requires the talents of professional human translators (programmers). However, competent a programmer may be, he is unlikely to have a complete grasp of all the fields in which he may be asked to function. The ideal solution to this problem is to devise a language acceptable both to computers and to the workers in a particular field.

The numerical language of computers cannot be changed much. They are dependent upon the structure of the particular machine itself. However, translation programs can be written. Thus, much attention has been and is being devoted to the design of application oriented languages.

Common programming languages, FORTRAN, ALGOL, COBOL, and others are not the answer. They have been too much

dominated by computer requirements. They are far removed from human requirements and the overall effect is that the essential points of algorithms described in these languages disappear in a mass of "housekeeping" details. In ALGOL, for example, a program which merely sets one matrix equal to another has twice as much space devoted to "housekeeping" as to essential details.

```
e.g.  for    i:=1  step 1  until n do
      for    j:=1  step 1  until m do
      A[i,j] := B[i,j];
```

Often the intended meaning of a series of statements is obscured by their complexity and interaction. This is particularly true when complex branching is needed to perform a particular task.

In addition to the difficulties within one discipline, there are others which arise out of the application of computers across the boundaries of disciplines. A single algorithm may contain processes from many fields. The description of this algorithm in an algorithmic language composed of the original notations for these fields may become ambiguous.

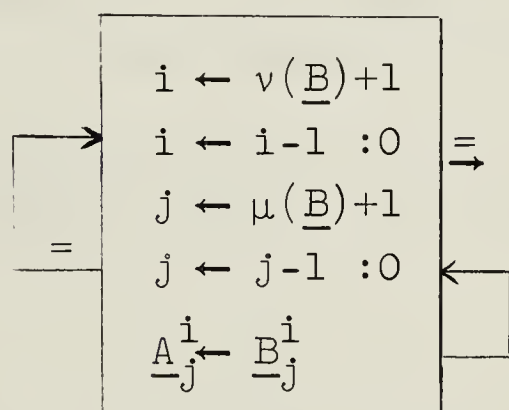
A programming language which attempts to be a common language for various disciplines must also have the property of being both precise and concise.

Section 1.2 Iverson's Language

In 1962, Iverson proposed a language which provides a new approach to the solution of the above problem. The language is both concise and precise and is closely related to common mathematical concepts and notations. It provides a uniform notation suitable for the description of algorithms which deal with sets, logical variables, vectors,

matrices, files, strings and trees. The language has been applied to such diverse fields as numerical analysis, computer hardware and software description, searching and sorting techniques and the physical representation of variables.

A program in this language is displayed as a vertical array of statements. Statements are normally performed in the specified order from top to bottom, but this order is alterable through the use of auxiliary arrows to show branching. Statements are of two types, specification and branch. The specification statements are distinguished by containing a left-pointing arrow as $x \leftarrow y$ which is read x is specified by y . The branch statements contain a colon which denotes a comparison between the objects separated in order to determine a possible branch. For example, the ALGOL program of section 1.1 may be rewritten in Iverson's language as:



(note that $\underline{A}_j^i$ is written A_{ij} in algebra)

A major part of the power of the notation lies in the suppression of details of indexing for entire or partial arrays. Thus, the above program (and hence the Algol program of section 1.1) may be more simply written as $\underline{A} \leftarrow \underline{B}$.

All operations, normally defined upon scalors are extended systematically on a component-by-component basis to

such arrays as vectors and matrices. Thus, for vectors $\underline{x}, \underline{y}$ and $\underline{z}$ and operation $\odot$ (e.g. +), $\underline{z} \leftarrow \underline{x} \odot \underline{y}$ implies that $\underline{z}_i$ is specified by $\underline{x}_i \odot \underline{y}_i$. $\odot$ includes all arithmetic, logical and relational operators.

The mere extension of normal operators to arrays would be incomplete in itself. The Iverson notation also includes a variety of selection operations which allow ready access to any part of an array. The simplest is the compression operation and is denoted by $\underline{u}/\underline{a}$ where $\underline{u}$ is a logical vector. This operation deletes from $\underline{a}$, those components corresponding to zeros in $\underline{u}$. More complex selections are readily done including mappings and permutations.

Another powerful feature provided is the reduction operation which is written as $\odot/\underline{a}$, where $\underline{a}$ is a vector and $\odot$ is any binary operator defined on the elements of $\underline{a}$. It is defined as meaning: $\underline{a}_1 \odot \underline{a}_2 \odot \dots$. Thus $+\underline{a}$ is the sum of all elements of the vector $\underline{a}$ and $\neq/\underline{u}$ forms the "exclusive or" of the elements of a logical vector $\underline{u}$.

The ordinary matrix product of matrices $\underline{X}$ and $\underline{Y}$ is written in ordinary algebra as $\underline{Z} = \underline{X}\underline{Y}$ and is understood to mean that

$$\underline{z}_j^i = \sum_{k=1}^n \underline{x}_k^i \times \underline{y}_j^k . \quad \text{This is written in the Iverson}$$

notation as

$$\underline{z}_j^i = +/(\underline{x}^i \times \underline{y}_j) \text{ and may be further shortened to } \underline{x} \overset{+}{\times} \underline{y} . \quad \text{This notation is generalized to } \underline{x} \overset{\odot}{\underset{2}{1}} \underline{y} .$$

In the field of symbolic logic, this notation has provided some rather startling results. For example, De-Morgans laws may be generalized and extended to vectors and written as $\underline{\wedge}/\underline{x} \longleftrightarrow \overline{\underline{V}/\underline{x}}$ and $\underline{\neq}/\underline{x} \longleftrightarrow \overline{\underline{=}/\underline{x}}$. These may then be extended to matrices and written in a single statement, viz.

$$\underline{x} \begin{matrix} \textcircled{\circ}^h \\ \textcircled{\circ}^k \end{matrix} \underline{y} \longleftrightarrow \overline{\underline{x} \begin{matrix} \textcircled{\circ}^h \\ \textcircled{\circ}^k \end{matrix} \underline{y}} \quad \text{where}$$

$\textcircled{\circ}$ represents the matrix of operators $\begin{pmatrix} \wedge & \vee \\ \neq & = \end{pmatrix}$.

The systematic extension of powerful operators to such other structures as trees and files provides a strong base from which to work in these fields. It suffices to say here that this notation unifies these fields also.

Section 1.3 The Iverson Notation in Logic Design

One particular area in which the Iverson notation has found ready acceptance is that of description of the logical structure of digital machines. Here, a register, "at the transfer-of-information level" is treated as a vector. A memory, which is a collection of registers, is readily described as a matrix. Transfers between registers, the selection of particular portions of a register and the various logical and arithmetic operations which may be performed upon the register are easily handled by the powerful operators of the language. Thus, $\underline{a} \leftarrow \underline{b}$ implies the transfer of the contents of register $\underline{b}$ to register $\underline{a}$. $\underline{a} \leftarrow \underline{a}^6/\underline{b}$ implies the transfer of the first six bits of register $\underline{b}$ to register $\underline{a}$ since $\underline{a}^6$ is the prefix vector (111111 0000 ...). $\underline{c} \leftarrow \underline{a} \wedge \underline{b}$ implies the transfer of the logical and of corresponding bits of registers $\underline{a}$ and $\underline{b}$ to register $\underline{c}$.

The Iverson notation has been used to great advantage in the description of the IBM 7090 [36], System/360 [20], and the IBM 1620 [1]. Falkoff has exhibited several different methods for handling the logical structure of associative memories in [19].

One immediate use of these descriptions is that they provide a mathematical description of the organization of a piece of hardware at a level which is understandable and useful to a user of the device. It enables him to determine precisely what will occur under a given set of circumstances. In doing this, it is not necessary to search through a computer users' manual which may not have anticipated these given circumstances. Furthermore, since these manuals use English as the main vehicle of description, the description if present, is not nearly so precise.

The Iverson notation provides a useful tool for the use of the system architect when planning and revising the structural organization of a digital machine. A description in this notation is a precise document which can be used to convey unambiguously this structure to the designers of the more detailed logical organization.

Section 1.4 Machine Simulation at a Logical Structure Level

Machine Simulation at a logical structure level consists of setting up a digital machine so that:

- 1) it has a series of elements which correspond to the registers of another machine.
- 2) it may be made to transfer information between these registers, in the same manner as the simulated machine.

The importance of simulation in digital engineering has been recognized for years. It is difficult to debug a piece of hardware once it has been built and packed into a small volume.

Also, hardware is now cheap and consequently the importance of minimizing circuits has decreased. More important is the time necessary for the design and checkout of new machines. Also important is the speed of the machine while performing a given task. In many instances the increase of this speed can occur only through a change of the logical structure. It is in these areas that simulation can help.

Another use of digital logic simulation occurs in the generation of diagnostic tests. These tests will be used by the serviceman to locate machine faults. It is desirable that they be simple and efficient. More important is the necessity that they do what they are supposed to do. A logic simulator which allows ready change of the machine organization can be used to simulate a machine fault and thus test these diagnostics.

Simulation of this type requires some vehicle for describing the logical structure. Very little has been done in the improving of this vehicle. An excessive amount of detail is often required and hence changes and debugging are difficult. Fortran IV has been used in [9]. This is a significant improvement, but still, unnecessary detail is required.

The ability to execute automatically, a language with the power of Iverson's language would provide an appropriate vehicle. The description of the machine by the system architect could be used directly to set up the simulating computer.

Section 1.5 Automatic Execution of Iverson's Language

The only previous attempt to provide a system for automatic execution of the Iverson notation is that of Hellernan [29]. It is an interpretive system and does not allow compound statements.

A pilot model of a compiler capable of performing simulation of the type described in the preceding section has been provided. It is restricted to a subset of the language

useful in this field. This compiler is described in the Appendices. In connection with this compiler, the problem of compiling techniques for the complete language has been investigated. It is the results of this which form the subject of this thesis.

CHAPTER II

Conventional Compiling Techniques

Section 2.1 Vehicles for the Description of Algorithms

The description of a particular algorithm can take on a variety of forms. One of the most obvious is in a language of everyday communication such as English (Fig. 2.1(a)). There are however, several disadvantages to this, not the least of which is a tendency to inexactness due to the use of words which have several meanings. The situation can be further complicated when the context does not resolve the ambiguity.

The machine code of a computer provides a vehicle for the description of an algorithm (Fig. 2.1(b)) which is unambiguous and exact but, unfortunately may have other disadvantages to a human user. Firstly, a series of numeric digits is not readily associated with any but the most trivial of processes. Another major disadvantage is the large fraction of information which is displayed explicitly because of a lack of implied meaning in a machine code (this of course means a large amount of printing for a small amount of information).

In order to avoid the first disadvantage above, symbolic information can be introduced giving what is referred to as an assembly language (Fig. 2.1(c)). Statements in an assembly language program have a direct one-to-one correspondence with statements in an equivalent machine code description. The second disadvantage is still retained and so symbolic languages, which allow compound statements have been introduced in vast array. (Fig. 2.1(d)). Compound statements are statements which, like ordinary algebra, can indicate a sequence of operations to be performed. Very few (if any) are sufficiently versatile to be able eliminate the second disadvantage in all instances. The description is still not sufficiently concise so as to allow a major portion of a process to be described in a single statement. A notation of the form proposed by Iverson [36]

with its introduction of a compact description of structured operands and a wide variety of operators is a significant improvement (Fig. 2.1(g)).

Often a more graphic form is desired. The flow chart or block diagram has long filled this need for computer programmers, engineers, management and others (Fig. 2.1(h)). It too can lack exactness since the description in the block is often merely a description in English. If the diagram is sufficiently detailed to amend this, the advantage of being able to grasp the entire process at once, is lost since for a process of any reasonable size, several sheets of paper would be required.

A graphical representation which has recently come into vogue is the tree (Fig. 2.1(i)). This form shows clearly the exact relationship of elements of a compound statement, but has not been extended to include a complete algorithm (branchings are a major stumbling block).

Both of these graphical methods are awkward for machine usage while in their usual form. The tree however, has several alternative descriptions which represent the structure exactly, concisely, and in form readily manageable by a computer. The one of major concern here is given the name left list matrix [36]. This consists of two vectors, $\underline{n}$ and $\underline{d}$. $\underline{n}_i$ is the i^{th} node and is ordered on increasing node numbers. $\underline{d}_i$ is the degree or branching ratio of node $\underline{n}_i$ (Fig. 2.1(j)). If the degree of each node (which in our usage is either an operator or an operand) is a known function, then $\underline{d}$ may be omitted. The resulting form (Fig. 2.1(k)) which consists of $\underline{n}$, the node vector alone, is sometimes given the name left list vector.

This form was used quite independently of any tree concepts by Lukasiewicz [50] as a normal form in formal logic.

It has since come into widespread usage in this discipline, and is known under a variety of names such as Lukasiewicz notation, parenthesis-free or Polish string. It has advantages over the ordinary algebraic notation of most high-level symbolic languages in that:

- 1) Parentheses are not required to indicate the order in which the operations are to be performed since there is only one possible order of execution.
- 2) Precedence of operators is neither required nor allowed.

a) ENGLISH

The value of each element of the vector a is to be calculated by multiplying the index of this element by the previous element for all elements other than the first. The first element is equal to 1.

b) MACHINE CODE (7040)

00000	0774	00	1	00001
00001	0774	00	2	77777
00002	0534	00	4	00021
00003	-0634	00	4	00007
00004	0560	00	0	00030
00005	-0600	00	2	00021
00006	1 00001		1	00007
00007	3 00000		1	00013
00010	0634	00	1	00027
00011	0200	00	0	00027
00012	1 77777		2	00005

c) ASSEMBLY LANGUAGE (MAP)

AXT	1,1
AXT	-1,2
LXA	N,4
SXD	TEST,4
LDQ	=1
STQ	A-1,2
TXI	*+1,1,1
TEST	TXH
TXH	EXIT,1,**
SXA	I,1
MPY	I
TXI	LOOP,2,-1

d) FORTRAN

```
A(1)=1
DO 10 I=2,N
10 A(I)=A(I-1)*I
```

e) ALGOL

```
a[1]:=1;
for i:=2 step 1 until n do
a[i]:=a[i-1] x i;
```

f) COBOL

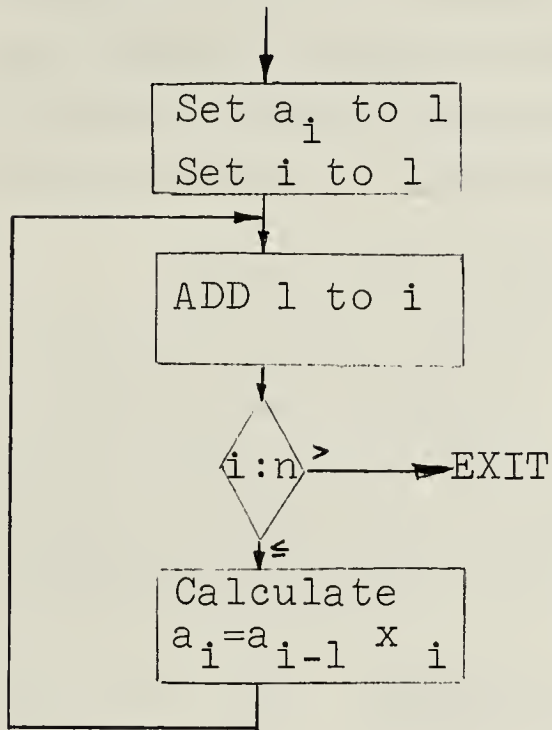
```
COMPUTE    A(I)=1.
PERFORM CALC VARYING I FROM 2 BY 1 UNTIL I IS GREATER THAN N.
CALC.  COMPUTE    I1=I-1
        COMPUTE    A(I)=A(I1)*I.
```

Fig. 2.1

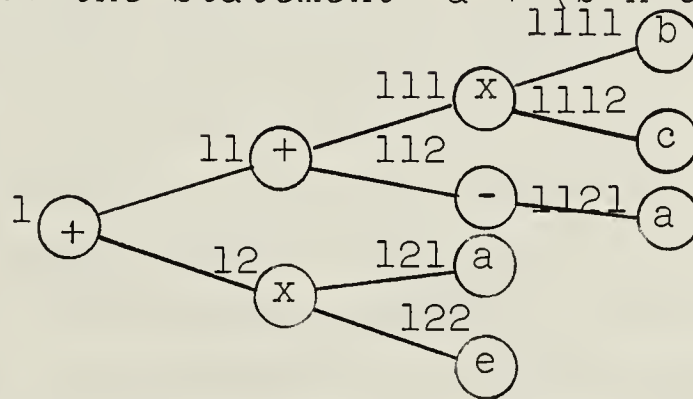
g) IVERSON'S LANGUAGE

$$\underline{a} \leftarrow x / \underline{n} / \underline{l} \quad (n)$$

h) FLOW CHART



i) TREE (for the statement $-a + (b \times c) + (e \times a)$)



j) LEFT LIST MATRIX

<u>d</u>	<u>n</u>
2	+
2	+
2	x
0	b
0	c
1	neg
0	a
2	x
0	a
0	e

k) LEFT LIST VECTOR OR LUKASIEWICZ STRING

(+,+,x,b,c,neg,a,x,a,e)

Fig. 2.1

An excellent description of the various forms this notation may take together with methods for translation to and from Lukasiewicz form has been provided by Hamblin [28]. It suffices for this discussion to note that Reverse Polish and Forward Polish differ in having the operator follow its operands in the former and proceed them in the latter. It is the Reverse Polish which is most commonly used in computers because, as will be seen later, it is a natural order to be used during the mechanical evaluation of a compound statement.

While this form is somewhat confusing to a human, a little practice renders it almost as understandable as the conventional notation. Unfortunately, many people are unwilling to submit to this practice and so the Lukasiewicz notation does not completely answer the need for a common notation for describing algorithms.

Section 2.2 Compilers and Compiling Techniques

Until such time as this common language is available, or until machines which will execute a high-level language directly are available [51], it will be necessary to translate between the various notations mentioned above. When this translation takes place from a high-level symbolic language to either an intermediate assembly language or directly to machine code, this process is called compilation and a program to perform such a process is called a compiler. This high-level input language is referred to as a source language and the language of the translated program is called a target language.

Some of the tasks of a compiler are:

- 1) Recognition of the various components of the source language
- 2) Handling of declarations of identifiers
- 3) Analysis of source statements including the translation of compound statements into a sequence of either machine codes or macro operations by examining the structure of parentheses.

Depending upon peculiarities of the source language, some of these may be more difficult than others. The detection of errors in construction and output of error messages must be part of any practical compiler since the majority of programs submitted to it will have errors. It is however, item 3 which has been subjected to major efforts in programming and analysis. Consequently, the most significant improvements have also occurred in this area. Some of these will now be described.

It is useful to split the various compiling techniques into two classes 1) Direct scans and 2) Syntactically oriented techniques. The first type have been used for almost all major compilers to date mainly because they are generally quite efficient and were used first in time. They have the drawback that they are tied closely to the source language thereby making source language changes difficult and in general they require more programming effort. The second type have been receiving much more attention by developers of compiling techniques because of the hope of reducing the cost of compiler writing while not incurring too great a loss of efficiency.

Section 2.3 Direct Methods

One of the first compilers, designed by Rutishauser in 1952, used an initial left to right scan in which level numbers were assigned to each operator, operand and parenthesis. Starting from 0, the level would increase by 1 for each "(or operand and decrease by 1 for each)" or operator. Then the statement was scanned repeatedly, each time reducing the highest level sub-expression to a single element.

The first Fortran compiler (described by Sheridan) replaced all constants and subscripted variables by simple variables (this is the reason for Fortran's restriction on subscripts) and then inserted additional parentheses so that all operator

precedences were indicated explicitly. Each parenthesized expression was then broken down into segments which could be split into triples consisting of an operation, operand and a level number. Redundant triples produced by unnecessary parentheses were then eliminated and the expression was optimized by recognizing equivalent triples.

Another early technique which has been used in several variations consists of scanning forward until a right parentheses is encountered and then scanning the sub expression between parentheses. This sub expression must be scanned once for each precedence level in the source language and is thus quite redundant in passes at the input string. When this is complete, the scan continues until the next right parentheses.

Section 2.3.1 Stacks

The stack principle was introduced in 1957 by Bauer and Samelson. It has since re-appeared under a variety of names such as push-down list, nesting store, yo-yo list, cellar, and last-in-first-out queue. A stack behaves in an analogous manner to a pile of plates in a cafeteria. There, a plate may be added to the pile or removed from the pile only at one position, the top. All other plates are inaccessible. Similarly, only the top or last item in a stack may be changed. Thus, a vector $\underline{p}$ behaves as a stack if only the following operations are performed on it.

- 1) $\underline{p} \leftarrow x, \underline{p}$
- 2) $x \leftarrow \underline{p}_1$
 $\underline{p} \leftarrow \underline{\bar{a}}^1 / \underline{p}$

The first operation is referred to as a push-down and the second as a push-up.

Section 2.3.2 The Stack in a Compiler

The use of a stack in a compiler can be explained with the use of Dijkstra's analogy to a railroad shunting yard in the form of a T-junction. This T-junction also is of the form of a stack since cars may enter (stack) and leave (unstack) the siding only at the switch at the leg of the T. While in the leg, the order of the cars will remain unchanged. In a compiler, it is operators and delimiters, not cars, which are stacked and unstacked. The input consists of the source language string and the target language is a Lukasiewicz string.

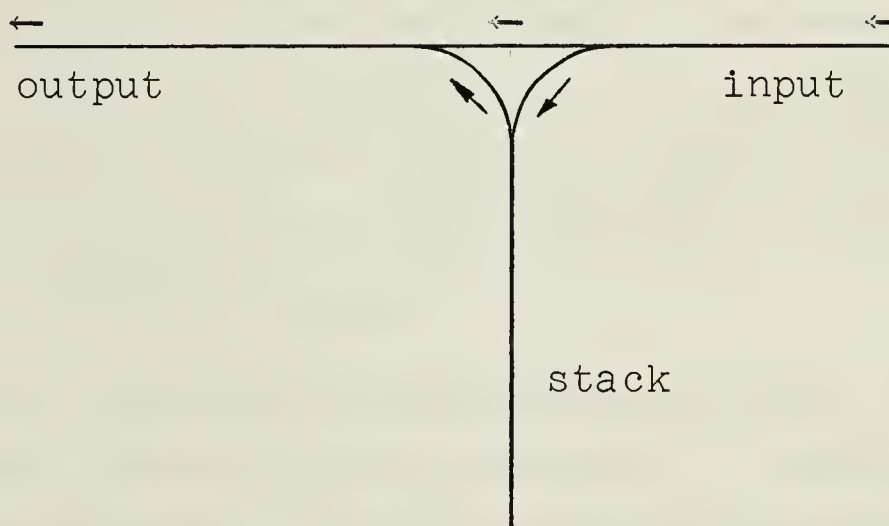


Figure 2.1

During translation, the elements move from input on the right to output on the left. Operands move directly across. Operators move via the stack and are not placed in the output until their operands have reached there. Delimiters such as parentheses are used to control stacking and unstacking. It is easy to build precedence rules into the decision of whether to stack or unstack.

Either a precedence vector or a decision table may be used to make this decision: the first, each operator (and

delimiter such as parentheses) are assigned a precedence. For example, a set of precedences suitable for translation of simple Fortran arithmetic expressions would be:

[	0
]	1
+, -	2
*, /, neg	3
**	4

"neg" indicates unary minus and replaces "-" whenever it appears to the left of [or at the beginning of an expression. These precedences would be stored in a vector $\underline{p}$ so that $\underline{p}_i$ is the precedence of operator i (under some convenient mapping)

e.g. $\underline{p} = (0, 1, 2, 2, 3, 3, 3, 4)$ if the mapping is $\underline{r} = ([,], +, -, *, /, \text{neg}, **)$ onto the interval vector $\underline{i}^0(8)$. Operator i is unstacked when operator j is the next element in the input string if $\underline{p}_i \geq \underline{p}_j$.

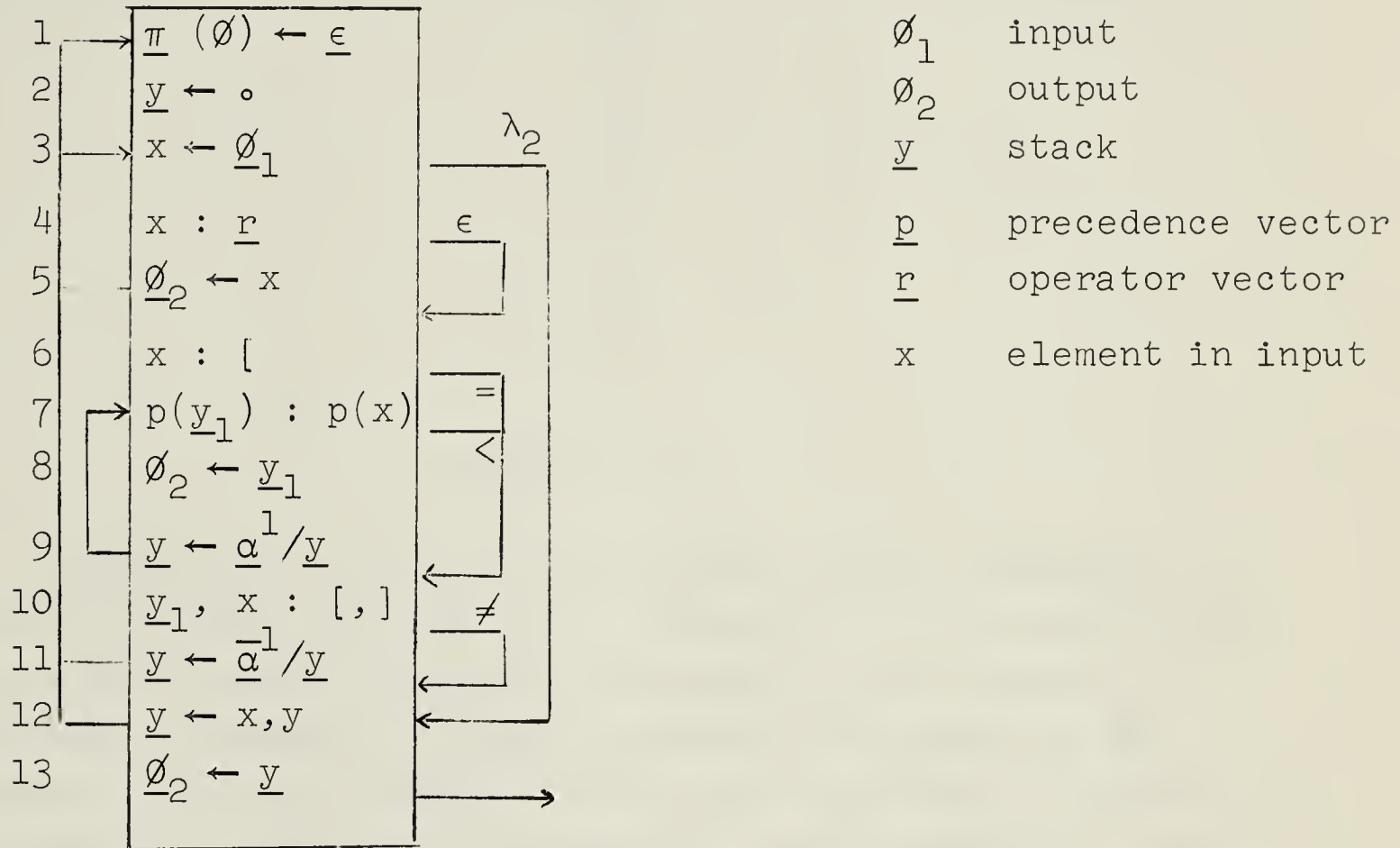
A few additional rules are necessary. First "[" can never cause unstacking and secondly "]" causes unstacking until the matching "[" is encountered at the top of the stack at which time, both are discarded. Finally, when the end of the statement is reached the entire stack is emptied.

The use of a decision table is really equivalent to the above. The table is stored as a logical matrix $\underline{T}$ which is determined by the property that if operator i at the top of the stack is to be unstacked when operator j is encountered then $\underline{T}_{ij}^1 = 1$ otherwise it has the value 0. It will be seen that $\underline{T}$ is the outer "product" of $\underline{p}$ with itself under the operation $\geq$ viz. $\underline{T} \leftarrow \underline{p} \overset{o}{\geq} \underline{p}$.

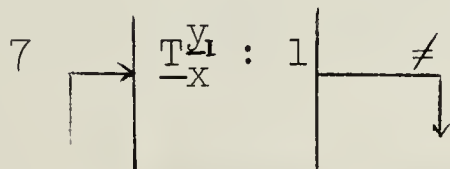
If $\underline{r}$ is augmented by the null operator $\circ$ with a precedence of -1 assigned to it, then program 2.1(a) will perform the

translation using the precedence vector method.

The replacement of step 7 in program 2.1(a) by the step 7 shown in program 2.1(b) will allow the use of the decision table in making the stacking /unstacking decision.



Program 2.1(a)



Program 2.1(b)

The functioning of this program will be described with the aid of a suitable example such as the expression $a+b*[c-d]$

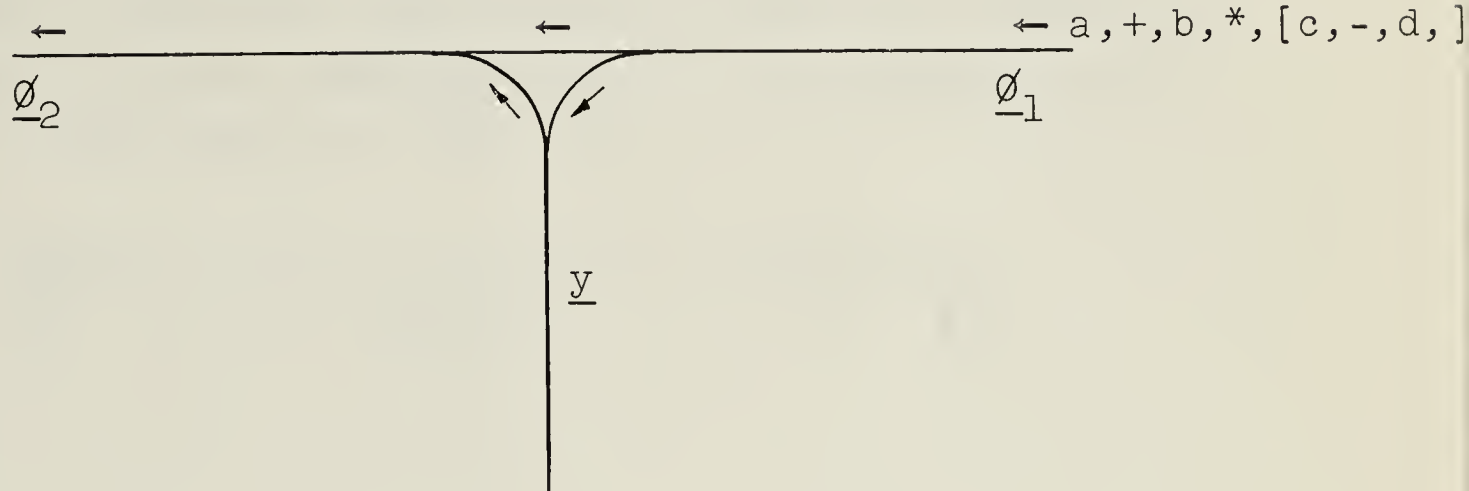


Figure 2.2

Initially, all files are rewound and the dimension of the stack vector y is set to 0. Element "a" is read from $\emptyset$, and since in step 4 it doesn't belong to $\underline{r}$ the vector of operators, it passes in step 5 directly to output in $\emptyset_2$. In step 3 the next element "+" is read and since in step 4 it does belong to the set of operators, and in step 6 it isn't " [", control passes to step 7 for the stacking/unstacking decision. The precedence of the null operator is -1 and that of "+" is 2 so the "+" is stacked in step 12 (in step 10 "o,+" is not the pair "[,] ") giving Fig. 2.3.

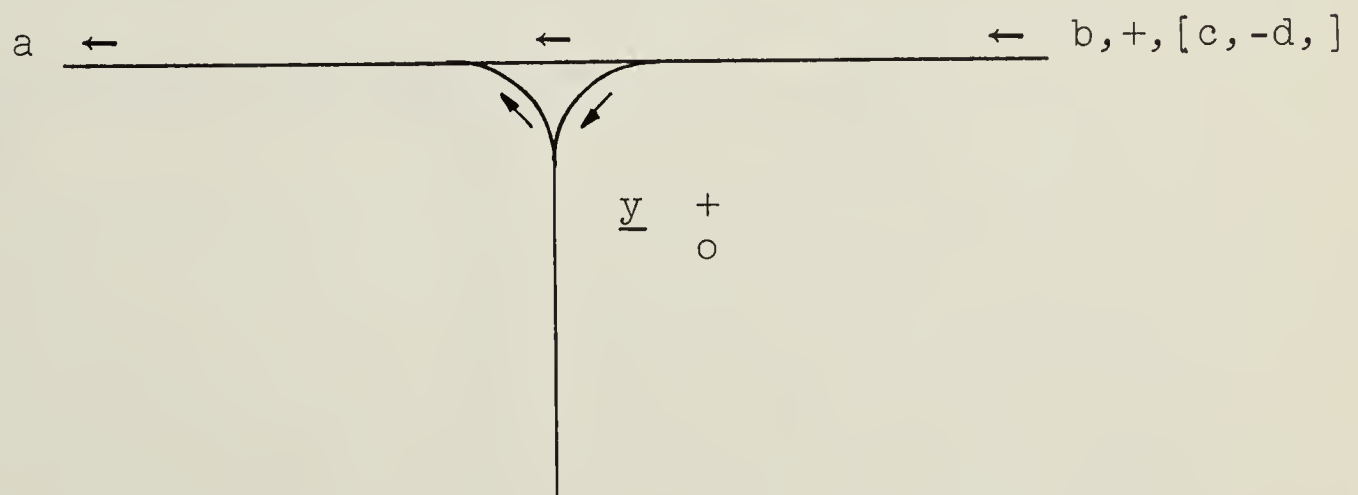


Figure 2.3

The next element read, "b" passes uneventfully, as did "a", directly to $\emptyset_2$. Following this, the "*" causes execution of the same steps as "+" and stacks on top of the "+" since $p("*") = 3$ and $p("+") = 2$. The " " which is encountered next forces a different route since at step 6, a branch to step 10 and 11 occurs whereupon the "[" is stacked on top of the *. The result is Fig. 2.4.

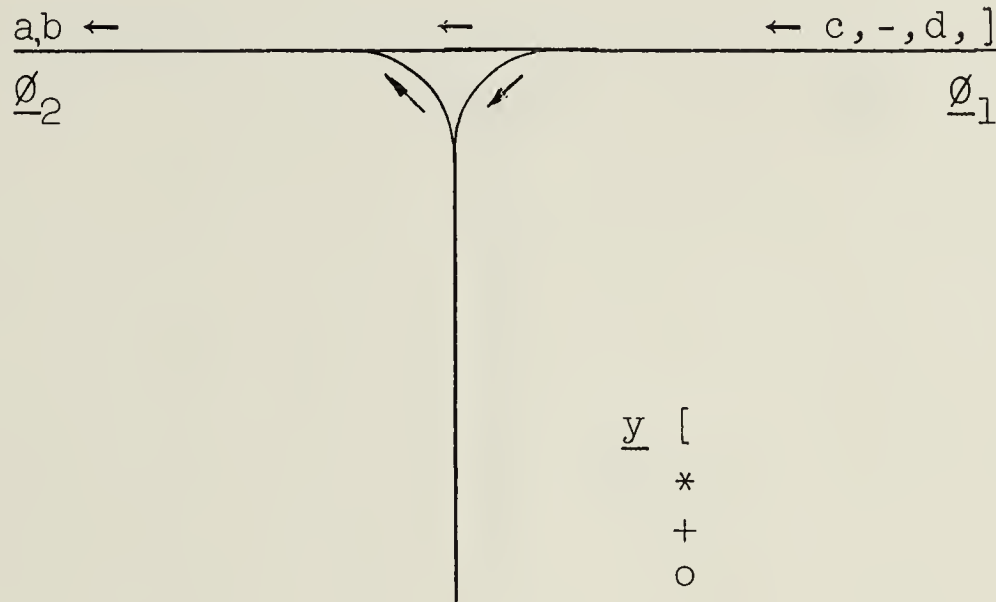


Fig. 2.4

Again, "c" passes directly to $\emptyset_2$, "-" is stacked above "[" since $p("-") = 2$ and $p("[")$ is 0. The last operand "d" moves to $\emptyset_2$ as usual giving Fig. 2.5.

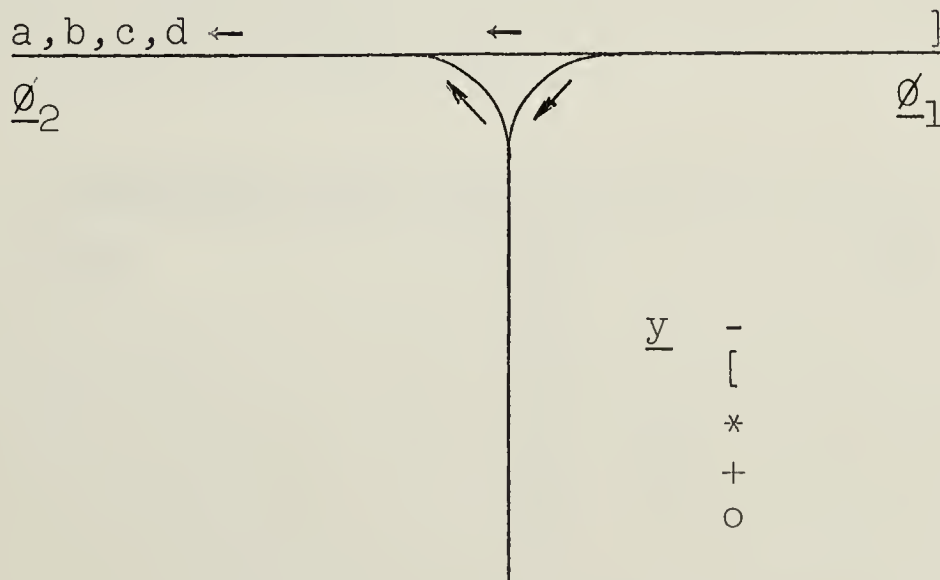


Fig. 2.5

When the "]" is encountered, the usual steps for an operator are followed until step 7. Here $p("-") = 2$ and $p("]") = 1$ causes unstacking of the "-" in steps 8 and 9 until "[" is at the top of y whereupon unstacking is discontinued. Then in step 10, it is recognized that the matching "[" has been encountered and both "[" and "]" are discarded in step 11. The result is shown in Fig. 2.6.

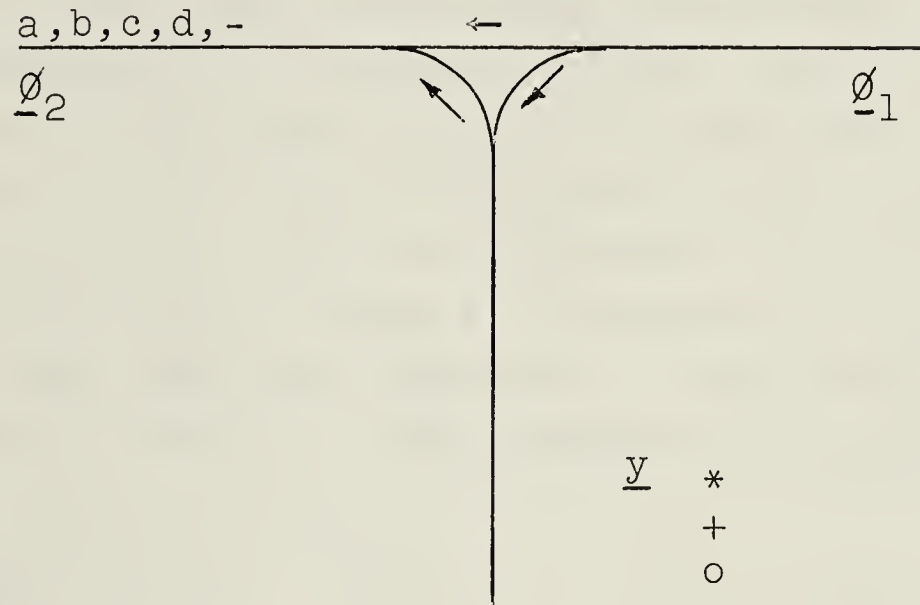


Fig. 2.6

When step 3 is executed, the partition symbol λ_2 is encountered at step 13, all of y is unstacked. At this point $\emptyset_2$ contains the statement in the form of a reverse Lukasiewicz string as shown in Fig. 2.7.

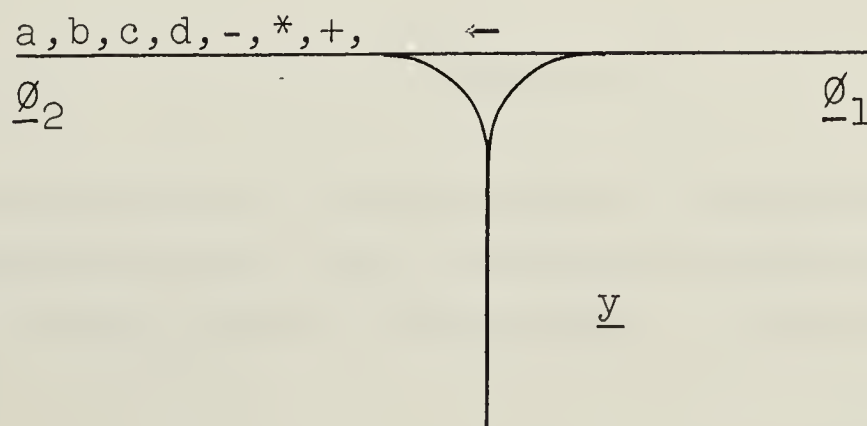
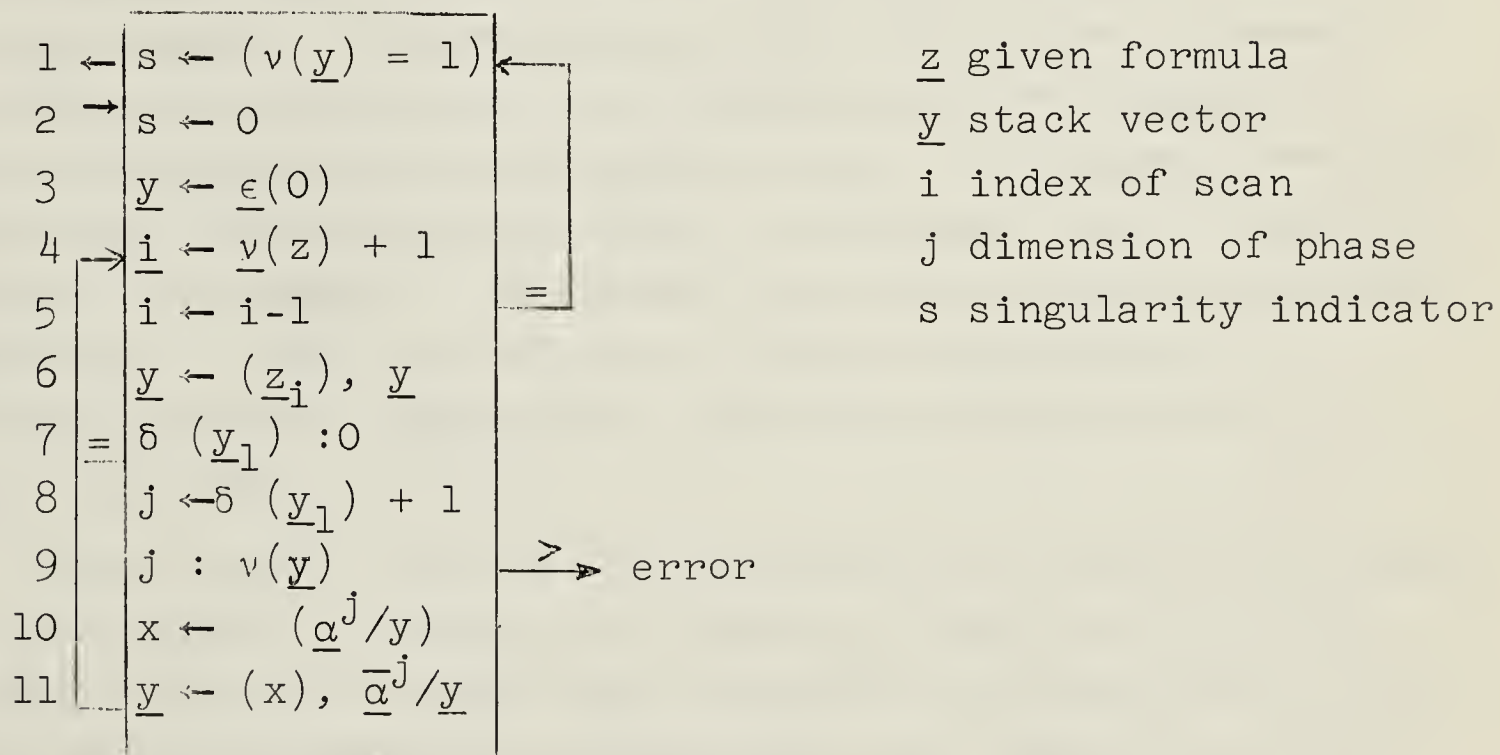


Fig. 2.7

Section 2.3.3 Evaluation of a Lukesiewicz String

For machines which require the string in this form (e.g., KDF9, B5000, ETL mk6), the process of compilation is completed. For others, it is necessary to further reduce it and for this process, a stack is also handy. If the statement is in Reverse Polish as in our case rather than the Forward Polish, then the statement is scanned from left to right, stacking all operands until an operator is encountered. At this point the operation may be "evaluated", its operands being those at the top of the stack. This evaluation is denoted by $(\underline{\alpha}^j/y)$. They are unstacked as evaluation takes place and the result operand is stacked in their place. The left to right scan then continues. This process is described in program 2.2 which is very similar to program 5.4 of [36].



Program 2.2

The process of evaluation of a Lukasiewicz string may be readily combined with the production of the Lukasiewicz string so that it never appears explicitly. This may be performed, by

executing the equivalent of steps 8 through 11 of program 2.2 whenever unstacking is about to take place in program 2.1 (steps 8 and 13).

Section 2.4 Syntactically Oriented Techniques

The key to the future of translation of mechanical languages seems to be in the area of syntactically oriented techniques. As mentioned previously, these techniques should allow rapid development of compilers thereby lessening their cost. As a side benefit, these compilers also necessitate a rigorous description of the language which should aid in eliminating ambiguities.

This rigorous description of the language syntax generally is done in a metalanguage such as the Backus Normal Form or Backus Naur Form (hereafter abbreviated BNF) used in the Algol Report. Unfortunately, while this gives a formal description of the syntax, the semantics of the language is usually described quite informally (e.g., the language described by only the syntax specifications of the Algol Report is quite different from Algol). An attempt to correct this by providing an addition to BNF which allows a formal description of semantics has been suggested by Irons and elaborated on by Ledley (see section 2.4.2).

There are two classes of syntactically oriented techniques. After the manner of Randell and Russell, they will be termed "Syntactic Routine Methods" and "Syntactic Information Methods". Of the two, the latter is considerably more powerful and more widely used.

Section 2.4.1 Syntactic Routine Methods

The translators in this classification are descendants of early translators which had a separate routine to handle each statement type. They have now evolved into a separate

routine for each syntactical unit. Thus, processors using these methods will consist of a set of subroutines, in either one-to-one or two-to-one correspondence with the set of syntactical units (plus a few housekeeping and control routines). Associated with each syntactical unit (hereafter abbreviated as SU) is a routine which is capable of recognizing that SU in the source string and another which will generate the desired code in the target language string. In some cases, these two functions may be paired into a single routine making the correspondence one-to-one.

For example, the language $\mathcal{L}_2\mathcal{A}_2$ described by Gorn [27] is defined in BNF by:

$$\langle \mathcal{L}_2\mathcal{A}_2 \rangle :: = x \mid (\langle \mathcal{L}_2\mathcal{A}_2 \rangle + \langle \mathcal{L}_2\mathcal{A}_2 \rangle).$$

A Syntactic Routine Processor for this language would consist of a routine which is capable of recognizing $\langle \mathcal{L}_2\mathcal{A}_2 \rangle$ and another routine which will generate the desired code corresponding to each $\langle \mathcal{L}_2\mathcal{A}_2 \rangle$ encountered. Clearly, for this example, the recognizing routine or recognizer must be capable of recursive call.

The control mechanism is essentially a list of routines currently attempting recognition plus associated parameters such as position in input string. This list is usually stored as a stack (push-down list) or a threaded list. Output from the generators is often in the form of a Lukasiewicz string rather than a machine or symbolic code.

These processors suffer from one of the same defects as those using a direct scan. Source language changes or additions will usually require a major reprogramming effort. On the other hand, since each of these recognizers is comparatively independent, reprogramming is less likely to introduce "bugs" with a direct scan.

A novel approach has been used by Leavenworth [47]. He shows that certain compiler languages can be used as meta-language for the description of some mechanical source languages (those for which BNF is usable). The properties required of the compiler language are relatively simple and exact. Most are possessed by common languages such as Fortran IV for the 7040/44/90/94. The few shortcomings of these processors may be readily by passed or added. An Algol compiler which performs resequencing of logical expressions would be even better.

The advantage of using a common compiler language as a metalanguage lies in the fact that upon compiling this description in the metalanguage, the control mechanism and recognizers are produced as a workable object program. Only the generators need be handled separately (calls to them are of course part of the control mechanism). Thus, the processor for the language $\langle \mathcal{L}_2 \mathcal{A}_2 \rangle$ above would be:

BOOLEAN procedure L2A2 ;

L2A2 : = x\g1\lparen\L2A2\plus\L2A2\rtparen\g2; plus, x, lparen and rtparen are basic identifiers for items on the input string. They advance the input pointer whenever recognition takes place. g1 and g2 are the generator routines and are called only upon recognition of the corresponding SU.

Ideas of this type seem to offer the best possibilities for applying Syntactic Routine Methods although conventional methods of programming these routines will probably continue to be used. More promising ways to obtain a syntax processor lie in the Syntactic Information Methods.

Section 2.4.2 Syntactic Information Methods

These processors are essentially independent of the language to be translated. They accept data on two levels.

- 1) The usual source language input string.
- 2) The syntax rules described in some metalanguage.

Although these rules may actually be loaded with the program they are still data since they exist at compile time as rules and not as recognizers.

Barnette and Futrelle [7] have provided a metalanguage called SHADOW together with a processor (running under IBM 7090 Fortran II) which enables automatic syntactic analysis of an input string whose syntax specifications are described in SHADOW. The output is a trace table which shows the position in the input string of each SU. Associated with these routines are another set of routines, the SASP routines which allow easy string handling operation. This is a good approach to the problem in that the resulting translator should be fairly efficient. Most inefficiencies could be eliminated by varying the SHADOW description.

Floyd [22] has provided a metalanguage which has provision for describing semantics also. A stack is used together with a set of "productions" which indicate tests and manipulations to be performed on the top of the stack. This language has been used by Evans as the basis for a processor. While the language itself is quite workable, the aforementioned processor is difficult to evaluate because of a lack of published results.

Brooker and Morris [59] have developed a "Compiler-Building System" which consist of routines to convert the meta description into a set of tables, routines to use these tables in syntactical analysis and a system of calling "format routines" or generators. The system is itself described in the metalanguage which it will process and is thus a compiler compiler.

Ingerman [31] has given procedures for the analysis of any language describable in BNF. They also use the meta description in a tabular form. The output is in a form which

can be easily modified to a Lukasiewicz string. No provision is made for including generators or any method of handling of the semantics.

The pioneering efforts of Irons [33,34] have been quite successful in this area. The meta description consists of a set of rules, each of which is composed of a syntax formula and a string of symbols designating the corresponding semantics. By way of illustrating Irons methods, Ledley [49] has used a novel example of translation of English into Korean which makes the whole process quite clear.

As pointed out by Irons [34] one of the main weaknesses of many syntactically oriented processors is that when a string which is syntactically incorrect is encountered, the information which would allow the generation of a meaningful error message is no longer available. Even worse is the problem of how to continue analysis of the rest of the string following this. The seriousness of this is soon evident when one realizes that the vast majority of programs presented to a compiler will have errors in them. Irons has exhibited an algorithm [34] which goes a long way towards overcoming both of these difficulties. In it, when more than one parse is possible for an input string, all are retained as possibilities until one is selected.

None of these syntactically oriented procedures make provision for optimization of object programs. Most have been excessively inefficient at execution time when compared with direct scans. Still, the use of these techniques allowed Irons to obtain a processor for Algol 60 within a few months of publication of the Algol 60 report.

CHAPTER III

Scan of an Iverson String

Section 3.1 Problems in the Scan of Iverson's Language

It is readily apparent that the Iverson notation has several features which are unique. Further examination reveals that many do not immediately yield to standard translation techniques. Most of these split into two groups:

1. those differences which result from the use of structured operands,
2. those which result from the great variety of operators available.

Section 3.1.1 Transliteration

The design of a good transliteration scheme is one of the first problems to be tackled in the implementation of a compiler for a notation such Iverson's. Of course, the scheme chosen is highly dependent upon the particular hardware to be used. Nevertheless all schemes have several properties in common. First is the need for readability. If a language is to be used as a working programming language, the programmer must be able to readily attach meaning to the string of symbols. This should occur without continual back reference as to which "shift" a particular character is in. Second, the scheme should be simple - otherwise, one becomes involved in too many rules and loses sight of the actual program.

In the printed language, the exact operand represented by a particular identifier is determined by its type face. Thus, for example, the letter x could represent the literal "x", a scalar, a vector, a matrix, a tree or a file depending upon whether it is printed in standard or bold face, italic or roman, upper or lower case. The set of all possible

combinations of these must be mapped by the transliteration scheme onto the much smaller set of characters which exist on a physical input/output device. One solution is to allow the particular form of operand to which an identifier refers to be declared somewhere in the program and then insist that it be used for none other elsewhere. This imposes a restriction which is undesirable in a complete implementation but is livable in a partial one. A partial solution is to determine the particular operand from context. That is the vector X can be distinguished from the matrix X by the appearance of an extra index, which may of course, be explicitly null. However, this is no help in distinguishing between literals and scalars.

In addition, several other qualities of a transliterations scheme are desirable but not essential. The scheme should be such that either a compiler or human user can readily distinguish between what is part of the transliteration scheme and what is actually part of the language's system. In this way, the language remains essentially unchanged. (Just the opposite occurred when Algol 58 was implemented in the form of MAD. Then, definite changes were introduced in the language).

It is important that symbols used in transliteration be both concise and mnemonic. If they are not concise, then the language is robbed of a valuable attribute - the inherent conciseness in it. If they are not mnemonic, then the advantage of resorting to a symbolic language (section 2.1) is lost.

Section 3.1.2 Structural Parameters

The presence of structured operands, vectors, arrays, trees, arrays of trees, files and arrays of files, gives rise to structural parameter. e.g., $v(\underline{x})$ $\mu(\underline{A})$ and $\mu(\underline{\emptyset})$ (for a complete list see summary S.2 in [36]). These must be recognized by a compiler as special symbols. The modification to Iverson's language introduced in [41] eases the problem. Under this

interpretation, these structural parameters are written as unary operators operating on structured operands. i.e., $v \underline{x}$, $\mu \underline{A}$. Once again, a transliteration problem arises in that the greek letter nu (ν) may represent several different parameters depending upon the type face in which it is printed.

Section 3.1.3 Special Operands

This notation requires that the compiler recognizes a number of identifiers as being built-in and representing special vectors or matrices. This is not particularly difficult unless a poor transliteration scheme is chosen. The particular method chosen for data representation (see Chapter IV) will have an influence upon the semantics of these special operands. It is entirely possible that they should be interpreted as calls to a standard subroutine which returns the value of the vector or matrix.

Section 3.1.4 Operators

The feature of Iverson's notation which is perhaps the largest obstacle to compilation is also the one which provides a good deal of the notation's power. This is the generalized interpretation of operators. In an orthodox notation, operators always appear between two operands if they are binary operators or to the left of an operand if they are unary. This sequence is interrupted only by parentheses to indicate order of execution. The Iverson notation is not nearly so restrictive.

Operators are allowed to appear adjacent to each other. This occurs in the case of the reduction operation ($\odot / \underline{x}$). It also occurs when operators are "doubled" to indicate the column extension of vector operations to matrices ($\underline{M}, \underline{A}$). This is complicated by the fact that "/" is used both in selection and reduction.

The notation for the maximum prefix (α / u) or suffix (ω / u) of a logical vector or matrix, for the ordering

operation of any vector ($\theta_j/\underline{x}$) or matrix and for the set selectors ($\sigma/\underline{a}$ and $\tau/\underline{a}$) is very similar to that of the reduction operation. Here, the character which precedes the slash is not a binary operator, rather the combination of this character and the slash is a unary operator. Again, these must be distinguished from other operations which also use a slash.

Operators are sometimes split and appear surrounding their operands. The best examples of this are the mesh ($\backslash \underline{a}, \underline{u}, \underline{b} \backslash$), mask ($/\underline{a}, \underline{u}, \underline{b}/$) floor ($\lfloor x \rfloor$) ceiling ($\lceil x \rceil$) and absolute value ($|x|$). The last 3 of these are unary operations. They may be rewritten in the form more common to unary operators viz., preceeding the corresponding operand. This suggested in [41] as: $\lfloor x$, $\lceil x$, $|x$.

This however, introduces an ambiguity since $|$ is used for both absolute value and residue. In most cases, there is no problem since one is a unary and the other is a binary operator. In the case of a statement like $\underline{M} || \underline{N}$, there are two possible interpretations.

1. the column extension of the residue operator
2. the row extension of the residue operator, operating upon the absolute value of N.

Three obvious ways exist to resolve this:

1. A different symbol could be used for one of these operators.
2. The column extension of the residue operator could be assumed unless parentheses are used to indicate explicitly that the other meaning is intended.
3. Find some method of scanning in which recognition will take place when absolute value is written $|x|$.

The third possibility merely avoids the problem (the notation $|x$ is desirable since it is of the form of all other unary operators). The first would be more general in that the

extra parenthesis are not required and no artificial rules are required. On the other hand, it would be a definite change to the language which would require further investigation. The second thus, is at present, the best choice.

The above operators, mesh and mask may be readily viewed as tertiary operators. There are several others which may also be viewed as tertiary operators. Examples of these are:

Mapping ($\underline{m} \int_j \underline{a}$), Representation ($r(n) \overline{T}j$) and Residue ($b|_j n$).

It would be desirable to have some uniform method of writing all of these. This uniform notation should take into account the fact that the expansion operation ($\underline{u} \backslash \underline{b}$) is a special case of mesh ($\backslash \underline{a}, \underline{u}, \underline{b} \backslash$) for which $\underline{a} = \overline{\underline{e}}$ and that compression is a special case of mask in which no selection takes place when $\underline{u}_i = 0$. It would be desirable if these special cases could be indicated by merely eliding one of the operands. One possible notation is that of Lukasiewicz but since this would require extreme changes to the language; this is rejected at present. A more acceptable one, suggested in [43] is that tertiary and higher degree operators be written between the first two operands. Any additional operands are listed to the right, delimited by semi-colons. The operand to be written on the left is the controlling variable unless it may be elided. Thus, mesh is written " $\underline{u} \backslash \underline{a}; \underline{b}$ " and mask " $\underline{u} / \underline{a}; \underline{b}$ ". Mesh becomes expansion for $\underline{a} = \overline{\underline{e}}$ by elision of " $\underline{a};$ ". The transition from mask to compression also takes place readily in the same manner. Residue is written " $\underline{b} | \underline{j}; \underline{n}$ " with " $\underline{j};$ " being elided when unnecessary. Representation is written " $\underline{n} \overline{T} \underline{r}; \underline{j}$ " with " $\underline{r};$ " being elided when the radix, r is 2. The alternative notation for mapping, " $\underline{m} \int_j \underline{a}$ " also falls into line as " $\underline{m} \int j; \underline{a}$ ".

The notation for "not" was originally proposed as an overscore but has since [41] been changed to "~" preceding its operand. This also is in line with the proposed uniform notation and thus is desirable. This notation extends logically to higher degree operators, and has a distinct advantage when a direct scan technique is applied.

Operators may appear catenated together as in vectors and matrices of operators [19]. It is difficult to adapt present methods of translation to handle this since operators may be operands and vice versa. Possible operations on these rather special operands are the selection and specification operations. The use of literals implies that an operator symbol may be used as an operator literal. The only way to distinguish between this operator literal and the operator itself is via the typeface (or its transliterated equivalent).

One way in which the Iverson notation differs from all others is in the notation for inverse operations - they appear to the left of the specification operator. In addition to this, selection operations are allowed on the left. This generalization allows a much neater description of many processes. It also forces interpretation of " $\leftarrow$ " as an operator and allows the same scan routines to be used on either side. The presence of operations to the left of the specification operator is readily dispensable in a partial implementation since it is always possible to write the equivalent "right hand only" statement at a minimal loss in neatness and conciseness. This option is definitely desirable however, and isn't particularly difficult to include.

The generalized matrix product is an item which places binary operators one above the other. The transliteration of this combination is likely to prove difficult. Even though, as stated at the first of this chapter, it is desirable to separate the function of recognition of a particular syntactical unit from translation of the transliteration scheme, the

generalized matrix product is one case where this will prove difficult.

In many instances, where the meaning is clear, controlling parameters or operands or even operators (e.g., multiplication) may be elided. It would be desirable to allow the elision of some of these. (such as where the explicit display of the above items is completely unnecessary). For example, the j parameter in $h|_j i$. One borderline case is that of elision of n , for special vectors, such as $\underline{w}^j(n)$, used in selection operations. In all these cases, the meaning is clear. The dimension of the selection vector is readily determined from one of the operands. Yet, often it may not be determined until run time. Thus, it may be necessary for a compiler to examine the context in which a particular item occurs to determine all necessary information about it.

The occurrence of a vector in a subscript is used to indicate the mapping or indexing of operations ($\underline{a}_k$). While this is again unorthodox, it should cause no undue difficulty in translation. If it does, the obvious solution is to require the use of the suggested alternative notation ($\underline{k} \int \underline{a}$).

Section 3.1.5 Indices

The notion of being able to refer to an entire structured operand by omitting indices necessitates that the compiler know when these have been omitted. To a large extent, this problem can be anticipated and avoided while designing the transliteration scheme but some portion of it will remain. This can be crucial if, for example, the same identifier is allowed to represent either a scalar, vector, matrix, tree or file depending upon the type face (or its transliterated equivalent). Clearly it is necessary to know whether the subscript is null or whether it simply doesn't exist for the particular operand.

Section 3.1.6 Declarations

The problem of declarations may be handled in two ways.

Firstly, artificial statements may be introduced which contain the necessary information. Something of the type allowed in FORTRAN IV or ALGOL would be satisfactory. The only disadvantage of this is the necessity to venture outside the language to achieve them.

A second method is dynamic storage allocation has been implemented. This method makes use of structural parameters, on the left of the specification operator. Thus $\underline{vx} \leftarrow 10$, would declare $\underline{x}$ as being a vector of dimension 10. It still remains to have some method for declaring an operand to be logical, integral, numerical or arbitrary. There appears to be no workable method of doing this and still remain inside the language. Artificial naming conventions such as were used in Fortran are possible, but these have been shown by experience to be undesirable. A simple statement of logical, integral, etc. followed by a name list is probably the only out. The transliteration scheme may, as previously noted, require some additional statement at the same time. Once progression has come this far, there seems no point in not going all the way to the first possibility mentioned above.

Section 3.2 Direct Scan of Iverson's Language

At first sight, it would appear that the efficient direct scan techniques mentioned in Chapter II would not be applicable to the Iverson notation since these techniques require a strict operand-operator-operand or operator-operand sequencing. Closer examination reveals that with a few modifications, these techniques are applicable to at least a subset of the language and may possibly be used on the full language. A right to left scan which is an assumed part of Iverson's language, is easily included by switching the precedence for ")" to zero and the precedence of "(" to 1.

The large number of operators precludes the use of a decision table to control unstacking. This is apparent when one realizes that even a small subset will require over 30 operators and hence 900 table entries. The fact that there is no inherent precedence between operators simplifies the task of translation.

The reduction operation could easily be a problem since it consists of two adjacent operators. If this is viewed and handled as a single unary operator when stacking and unstacking, then the stack technique will readily apply. A similar treatment would be used for such operations as maximum prefix etc. ($\alpha/\underline{a}$).

For use in a direct scan, it is definitely desirable that "split operators" such as mask and floor be written in the form suggested above. When this is done, the unary operators are handled easily. The operators which are written as tertiary operators are handled by assigning a precedence to the operand delimiter, ";" which is lower than other operators (but of course higher than the specification operator). The fact that the same symbol is used to denote mask as for compression (i.e., "/") and to denote mesh as for expansion (i.e., "\") means that an additional feature must be incorporated to detect which member of each pair is intended. This must be done before the Lukasiewicz string is obtained and is most easily done at the moment of unstacking. When "/" or "\" is being unstacked, the next item in the stack should be checked to determine if it is the operand delimiter, ";". If so, then the degree of this particular selection operator should be advanced from 2 to 3. This scheme is dependent upon the low precedence of ";" to ensure that when a corresponding ";" exists it will be the next stack item. This scheme can be easily extended to quaternary and higher degree operators.

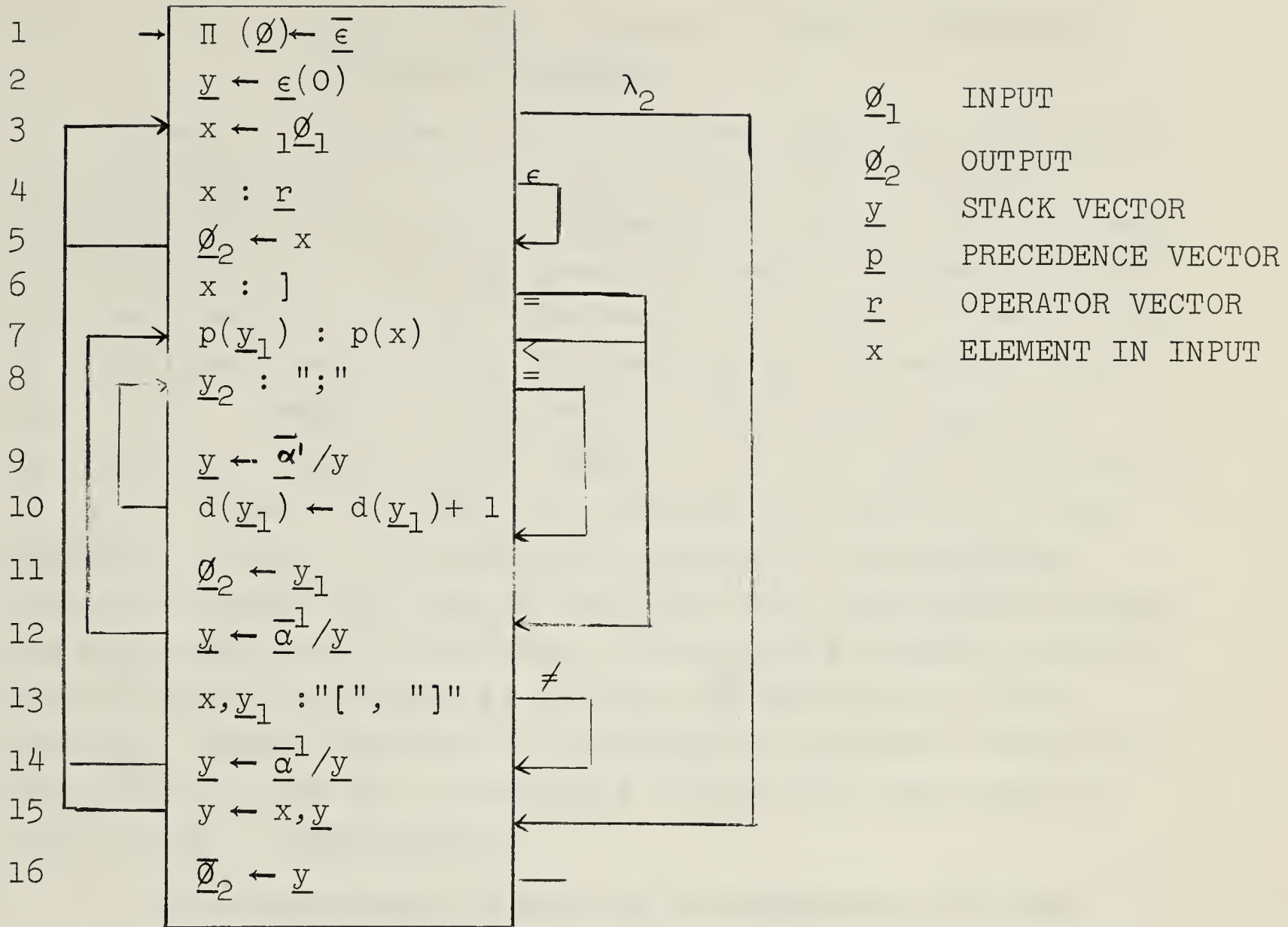
The specification operator " $\leftarrow$ " is assigned a precedence which is lower than all other operators but above "(" . Thus

it would be assigned a precedence of 2. This also allows the same routines to be used to scan the expressions on both sides of the specification operator.

As mentioned above, the recognition of the generalized matrix product is likely to be highly dependent upon the transliteration scheme. Once this recognition has taken place, the generalized matrix product is handled in the same manner as any other binary operator.

The above features are incorporated in Program 3.1. The detection of the reduction operation is not shown explicitly. This would occur during the execution of step 3 or 4 using a look-ahead type of operation. If an initial edit pass is taken to interpret the transliteration scheme, then the detection of the reduction operation could most easily occur here.

The significant changes from Program 2.1(a) incorporated in Program 3.1 occur in steps 8-10. Other changes occur in steps 1,3,6 and 13.



Program 3.1

)	Ø
(	1
←	2
;	3
all other operators	4

Table 3.1 Values assigned to precedence vector

Section 3.2.1 The Use of Syntactically Oriented Techniques on Iverson's Language

The use of syntactically oriented techniques provides the best possibility of obtaining a relatively complete implementation of Iverson's language. A decision must be made as to just which of the two possible forms of a syntactically oriented compiler to use. The result of Chapter II would seem to imply that the syntactic information method would be the one to use. Indeed, the generality and supposed ease of extending the implementation makes this type rather attractive. It is of course, impossible to describe the printed Iverson language in any of the presently available metalanguages. The major reason for this is the fact that these metalanguages are applicable only to strings. The printed Iverson notation uses position vertically as well as horizontally to give meaning. Transliteration to a string is therefore necessary. Even then, it may not be possible to describe the complete notation in a metalanguage.

Other weaknesses of present metalanguages are highlighted by the fact that most are not even powerful enough to describe present compiler languages. Algol for example, is not completely describable in BNF although a language much like Algol could be described in BNF. As pointed out in [12], the syntax description of Basic Fortran in BNF requires additional metaoperators and conventions (i.e., the metalanguage necessary is no longer BNF).

Most presently available syntax processors on the other hand require that the metalanguage used be a fairly simple one. BNF is among the most powerful ones handled. Thus, while syntactic information processors are attractive and for a transliterated subset are even practical, they have not yet reached the state of being useful for a complete implementation.

The syntactic routine methods must be resorted to and in this regard, some of the work of Leavenworth [47] comes to mind since it is actually a cross between the two types. The end result is a syntactic routine processor, but it is programmed as though Fortran IV was the metalanguage. It still retains the necessity of description in a metalanguage and Fortran IV is no better for the syntax description than any of the others.

Thus, the syntactic routine processor would have to be written as a series of recognizers each capable of invoking a one or more generators whenever recognition takes place. If these are kept quite separate, then the possibility of easy change and addition are retained. The summary of notation in [36] provides an excellent basis for the programming of these routines. If the compiler is written so as to make an initial edit of the input string prior to invoking the syntax routines, then even the transliteration details may be suppressed as far as these routines are concerned.

Section 3.3 Generators

A compiler for a conventional language will usually produce a more efficient object program if it compiles "in-line code". That is, to say that the main sequence is left only when a function or subroutine call is explicitly indicated in the source program. This is not the case for a compiler for a language such as Iverson's which has a wide variety of possible operations upon structured operands. The running time will, of course, be slightly longer if subroutines are called to perform most operations. The usage of storage will be more efficient if calls to subroutines are used. This is because in general, a single operation will result in more code if the operation is in an Iverson string than in the string of a lower level language such as FORTRAN and ALGOL. Another feature which heightens the need for run time subroutines is the fact that dynamic storage allocation of memory for intermediate results

is definitely desirable (see Chapter IV). This means that calls to subroutines to allocate this storage would also have to be included in line if run time subroutines for operations were not used. Consequently, the end product of the compiler should be a series of subroutine calls.

CHAPTER IV

Machine Organization at Object Time

Section 4.1 Dynamic Storage Allocation

Dynamic storage allocation means that the assignment of storage space occurs at object time and hence may be changed during execution. There are several levels at which this allocation may be effected.

Algol provides a rather simple form of dynamic storage allocation in that only the assignment of storage for arrays are changed [53]. Furthermore, this may occur only when a block is entered. So long as control remains within this block, the storage assignments of arrays local to this block remain unchanged. Once control leaves this block, this storage becomes available for use by other arrays which are local to a parallel block.

Quite a different form of dynamic storage allocation occurs in a system organization of the type found in Operating System/360. Here, the storage assignments for both program and data may be changed any time during the execution of the program. The storage assigned to a subroutine, may be reassigned to another routine or piece of data if reference is no longer being made to this routine. Similarly, reallocation of the storage assigned to an array could occur whenever this array was no longer being referenced. Particular blocks of information are moved between high speed and low speed memory as the need for them changes. This movement need not wait until a particular point in the program is reached.

A different form again is implied by various features of Iverson's language. The interpretation of the unary operators v and μ when they appear in an expression to the left of the specification operator is that the dimension of the structured operand to which this structural operator refers is respecified.

For example, the statement $v_x \leftarrow 10$ would change the dimension of the vector $\underline{x}$ to 10 and the statement $v_x \leftarrow v_x + 1$ would increase the dimension of $\underline{x}$ by 1. Thus, the reallocation of storage for structural operands may occur at any time during execution. The stack vector of program 2.1 provides a good example of a structured operand whose dimension may be varied at will.

Clearly the two operands of the specification operator must be of like dimension at the time of execution for the specification operator to have meaning. If the dimensions are not the same, meaning can still be attached to this operation if dynamic storage allocation is allowed. The most reasonable interpretation, then, is to assume that the dimensions of the operand to the left of the specification operator should be changed so as to be compatible with the dimensions of the operand on the right. For example, the statement $\underline{z} \leftarrow \underline{y}$ where $(v_z) \neq (v_y)$ would imply that v_z would be changed to the value of v_y prior to attempting specification. A more complex example will illustrate some of the complications which can arise. Consider the statement $\underline{x}, \underline{y} \leftarrow \underline{z}$ where $v(\underline{x}, \underline{y}) \neq v(\underline{z})$. Here there exists a dilemma of whether to vary the dimension of $\underline{x}$ or $\underline{y}$. The existence of a simple convention like "the change of dimension is to occur at the point of highest index" would solve this since $\underline{y}$ would be the one to be varied.

A third feature which implies dynamic storage allocation is the fact that compound expressions involving structured operands are allowed. In the course of evaluating an expression, temporary storage will be required for intermediate results. The dimension of these intermediate results are not the same throughout the expression and if their storage is not reallocated dynamically the following situation can easily arise. There will not be enough storage available to contain all of these intermediate results in spite of the existence of storage assigned to intermediate results which are no longer needed.

Section 4.2 Organization for Selection Operations

The selection operators are a very special group of operators in that they do not require that new values be created. Rather, the result is merely a rearrangement, perhaps with some omissions, of values which are already stored. Similarly, the set, rotation, mapping and permutation operators do not create new values. It is only when the specification operation is executed that information in storage is changed. That is, the statement $\underline{x} \leftarrow \underline{\mu}/\underline{y}$ creates no new results while the portion $\underline{\mu}/\underline{y}$ is being executed. It is only when $\underline{x} \leftarrow (\underline{\mu}/\underline{y})$ is being performed that the contents of storage need be changed. Conversely, a statement like $\underline{x} \leftarrow \underline{y}+\underline{z}$ does create new results when the operation $\underline{y}+\underline{z}$ is executed.

The representation of information should allow the program to merely keep track of where elements are stored. Thus, whenever new elements are not created (e.g., selection operations), it would be unnecessary to use an excessive amount of storage for elements which are already present. An operation like $\underline{\alpha}^1/\underline{x}$, which deletes the first element of the vector $\underline{x}$, is a good example of the type of operation which would otherwise almost double the amount of storage needed.

Section 4.3 Representation of Data

For a particular computer, the coding scheme will be fixed and hence investigation of this aspect will be rather fruitless for our purposes here. The physical devices used for storage may be numbered sequentially and hence may be considered as a vector. For example, the memory of the IBM 7040 may be considered to be a vector of 32768 elements each capable of storing a 35 bit signed binary number or a vector of 196,608 elements, each capable of storing one of 64 binary coded decimal characters or even a logical vector of 1,179,648 elements. It is of course more easily considered as the vector of dimension 32,768. Throughout the remainder of this chapter,

any such physical vector will be denoted by $\underline{\pi}$.

The representation of a particular quantity, is also a vector (e.g., a word). If this representation uses a continuous sequence of elements from $\underline{\pi}$ (an infix), then this representation is solid. For example, the representation of a quantity x by the contents of word 1432 would be solid but the representation of the same quantity by bits 3 to 17 of word 1120 and bits 11 to 29 of word 1356 would not be solid. Normally, the representation of a particular element is solid but it is not uncommon to find that the representation of a structured operand which is composed of these elements is not solid.

Throughout the majority of the succeeding subsections, discussion of allocation will be centred upon vectors since more complex structured operands can be reduced to an equivalent vector. Problems which may be encountered in this regard are discussed at the end of the chapter.

Section 4.3.1 Linear Representations

The simplest way in which storage may be allocated to a vector, $\underline{x}$ is to assign successive elements of $\underline{x}$ to successive elements of $\underline{\pi}$. For example, on the 7040, each element of a vector would be assigned to a word. Words corresponding to adjacent elements would also be adjacent. A slightly more general method is to allocate component $\underline{x}_i$ to $\underline{\pi}_j$ where $j=a+bi$, a and b constant. A representation in this form is termed Linear. Again, on the 7040, Fortran IV complex quantities are stored in adjacent words, the real part in the first word and the imaginary part in the second. A vector of complex quantities would be allocated to a series of adjacent double word entries which are infixes of the physical vector $\underline{\pi}$. The vector $\underline{x}$ consisting of all the real parts of the complex vector would be stored in a linear representation for which $b=2$. For the situation mentioned above wherein successive elements of $\underline{x}$ are allocated to successive elements of $\underline{\pi}$, $b=1$.

In this case, the linear representation of the vector is also solid.

Most common compilers generate linear representations of data. Even many Algol compilers are able to do this since the dynamic storage allocation occur only at specified points throughout the program and once storage has been allocated, this allocation is not changed until a block is either entered or left.

Section 4.3.2 The Explicit Grid Matrix

Often a linear representation is either inconvenient or inefficient. (e.g., if a vector needs to be changed by insertion or deletion of an element or when the size of the elements vary widely). An explicit grid matrix offers a useful alternative. The use of the grid matrix presumes that each element of $\underline{x}_i$ is solid in $\underline{\pi}$. The grid matrix, $\underline{\Gamma}(\underline{x})$ is a two column table with each row corresponding to an element of $\underline{x}$. The elements in the first column, $\underline{\Gamma}_1^i(\underline{x})$ give the leading address in $\underline{\pi}$ of the corresponding element of $\underline{x}, \underline{x}_i$. Elements of the second column, $\underline{\Gamma}_2^i(\underline{x})$ give the dimension of $\underline{x}_i$ in $\underline{\pi}$. If for example, the vector $\underline{x}$ has dimension 3, the first element occupying 3 words starting at address 199, the second occupying 1 word starting at address 003 and the third occupying 3 words starting at address 101 then the corresponding grid matrix, $\underline{\Gamma}(\underline{x})$ would be:

$$\begin{pmatrix} 199 & 3 \\ 003 & 1 \\ 101 & 3 \end{pmatrix}$$

Variations of this scheme include:

- a) storing the final address of $\underline{x}_i$ in $\underline{\Gamma}_1^i$ and the length in $\underline{\Gamma}_2^i$,
- b) storing the starting address of $\underline{x}_i$ in $\underline{\Gamma}_1^i$ and the final address in $\underline{\Gamma}_2^i$.

One advantage of the grid matrix is that elements may be inserted or deleted from the vector $\underline{x}$, merely by altering the grid matrix. In the above example, an element of 4 words starting at address 65 could be inserted between $\underline{x}_2$ and $\underline{x}_3$ (pushing $\underline{x}_3$ to a new $\underline{x}_4$ thus increasing the dimension of $\underline{x}$) by merely changing the grid matrix to:

$$\begin{pmatrix} 199 & 3 \\ 003 & 1 \\ 065 & 4 \\ 101 & 3 \end{pmatrix}$$

The location of the elements themselves would be unchanged.

It allows efficient use of storage if the length in π of elements of $\underline{x}$ vary greatly. That is, the loss of storage due to the explicit use of the grid matrix is easily over-balanced by the saving which results from the use of variable amounts of storage for each element. However, the storage occupied by the grid matrix is wasted storage, if the length in π of elements of $\underline{x}$ is uniform or if the longest one is quite short.

The grid matrix itself is usually stored in a linear fashion. Occasionally, it is advantageous to use some other representation for Γ . A common one is the chained representation described in the next section.

Section 4.3.3 Chained Representation

A chained representation incorporates the grid matrix in the representation of $\underline{x}$ itself. There are many varieties, but the essential features are that each element $\underline{x}_i$ has stored along with it, usually preceding, the length of $\underline{x}_i$ and the starting address of at least one other element of $\underline{x}$. If this other element is $\underline{x}_{i+1}$, then the representation is a forward chain. Similarly, if it is $\underline{x}_{i-1}$ then a backward chain results.

If both are present, then the representation is a double chain. A chained representation is frequently referred to elsewhere as list. In some instances, the length of x_i in π is constant for all i . In this case, the length may be omitted from the chaining.

Section 4.4 Pools

Implicit in the notion of dynamic storage allocation is the necessity of assuring that segments of storage are not assigned to more than one function at a time. A related task is the keeping track of which segments are available for use and which one should be assigned next. All the available storage can be organized into a pool. A change is made to the pool whenever storage is allocated or released.

The term queue of storage will be used to refer to the order in which the segments of storage in the pool are to be allocated.

There are several ways of organizing this pool:

a) the pool is organized as a stack (see Chapter II). Thus, the queue of available storage locations obeys a last-in-first-out discipline and, the most recently used storage component will be the next one assigned.

b) the queue discipline of first-in-first-out may be used. This results in cyclic usage of storage. Every storage component will be used once, before any component is used a second time. This type tends to give larger continuous sections of storage than that described in a) above.

c) storage may be assigned in such a way that the largest unassigned segment of π available is the next to be assigned. This produces a further advantage over b) in that even longer sections result. This scheme necessitates some method of keeping track of the lengths of all available storage segments.

d) one of the simplest ways is to assign storage in the order of increasing address among those elements of π which are unassigned. If the storage is being assigned to elements of a vector, then it might be desirable to obtain adjacent storage locations for adjacent elements of the vector. This is easy with this scheme.

Section 4.5 Representation of the Structured Operands of Iverson's Notation

The operations of selection and dimension changing prove to be quite tractable when the vector operands are represented by variations on the grid matrix scheme described in 4.3.2.

In contrast, a linear representation requires recopying, without change, portions of the original vector. Section 4.3 has detailed reasons why this should be avoided. Furthermore, whenever the dimension of the vector is increased, the following location may not be available. Hence a great deal of inconvenience could result from the use of a linear representation.

The use of a chained representation would require that either the chaining information be temporarily changed or that the appropriate chain segments be completely recopied whenever intermediate results from a selection operation are generated.

A grid matrix scheme eliminates both of these problems. It is easy to alter the dimension by minor changes to the grid matrix. The intermediate result of a selection operation is merely a new grid matrix while the original information remains unchanged.

A more subtle advantage of the grid matrix scheme derives from the fact that the selection operations have the same effect on either side of the specification operator. On both sides, it implies the creation of a new grid matrix and so, the same routines may be used for both.

There are a few useful modifications which may be made to the scheme as described in 4.3.2. If the elements of $\underline{x}$ are all the same length, and if this length is constant for a particular type of vector, then the elements of $\underline{\Gamma}_2$ will all be the same and will be dependent only upon the type of vector. In this case, providing the vector type is known at any point in the program (perhaps a table), the vector may be stored piecewise. This means that whenever a segment of $\underline{x}$ is assigned memory locations in a linear fashion, only a single row of the grid matrix is used. The contents of $\underline{\Gamma}_1^i(x)$ is the first address of the i^{th} linear segment of x and $\underline{\Gamma}_2^i(x)$ is the number of elements of $\underline{x}$ in this segment. Thus, if the entire representation of $\underline{x}$ is linear, then the grid matrix consists of a single row.

Because the size of the grid matrix itself may vary it is convenient to allocate storage for it dynamically. Thus, instead of a linear representation, a chained representation should be used for the grid matrix.

If the grid matrix is stored as a row list ($\underline{E}/\underline{\Gamma}$) then the length portion of the chaining information is redundant and may be omitted. Furthermore, since many segments of the grid matrix are likely to be stored in a linear position, it is advantageous to include the address of the next component only when the next component does not fit a linear scheme. The presence of this chaining information can be readily indicated by a single bit associated with each grid matrix entry.

A few problems are encountered when this scheme is applied to the equivalent vector representation of a more complex structured operand such as a tree. Here the execution of an operation peculiar to trees may require so many extra manipulations that it might be easier to use a chained representation of a tree directly. Similarly, column operations

on a matrix stored as a vector in a row-by-row fashion may present some difficulty, when the above scheme is used. Identities such as those suggested in [41] may be helpful to convert the operation to an equivalent operation which is more easily performed on a row list though this may not always be possible. An example of this is the identity $(A_{\sigma}^{\rho} B)_{\underline{q}}^{\underline{p}} \longleftrightarrow A_{\underline{r}}^{\underline{p}} \rho_{\sigma} B_{\underline{q}}^{\underline{r}}$ where ρ and σ are dual operators such as $\vee$ and $\wedge$. Here, the use of this identity will eliminate the need for a column scan of B.

Thus, variations of an explicit grid matrix allow efficient use of storage for most of the operations on the structured operands allowed in Iverson's language. A description of a scheme to be used on the 7040 is described in Appendix E.

CHAPTER V

Concluding Remarks

Section 5.1 Review of the Thesis

In the preceding chapters, we reviewed several aspects of a programming language proposed by Iverson. In particular its application to the logical description of digital machines has been presented, together its use in simulation if it possible to automatically execute programs written in this language.

A description of the compiling process has been provided. Several compiling techniques for an algebraic language have been discussed in some detail and other variations have been mentioned. Some problems encountered in applying these techniques to the Iverson notation have been presented. Methods for overcoming some of these have been developed and presented together with some of their limitations. Desirable features of a transliteration scheme are also included.

Several features of Iverson's language require an uncommon organization of the computer during execution. These features together with several methods of organizing the computer were enumerated and a suggested organization was presented.

The application of the above techniques was carried out in a "pilot model" compiler, written for the IBM 7040. For this pilot model, the source language was restricted to that subset useful for describing the logical structure of digital machines. The description of this pilot model has been relegated to the Appendices. At this point, it should be mentioned that a good deal of the investigation carried out in this thesis consists of programming and trying several methods of compiling the Iverson notation. Only the most successful one was included.

Section 5.2 Future Investigations

The wide applicability of Iverson's notation to finite sequential processes soon convinces one of the need for being able to execute directly a larger portion of the language than is required for use in the logical design of digital machines. This was illustrated quite aptly in Chapter II. There, the existence of a compiler for Iverson's Language, which is capable of handling variables of the arbitrary type would make the writing of a compiler much easier. Numerous examples have been exhibited which show the simplifications possible through the use of Iverson's Language in the field of numerical analysis. The only one which will be quoted here comes from [41] and expresses concisely the ordinary inverse of the matrix $\underline{A}$. It is: $(\underline{x})^{-1}/\underline{A}$. The notation provided for trees has been applied to information retrieval in [60] and to sorting in [36]. In both cases, a compiler for Iverson's notation which handles trees would simplify the programming of algorithms in this field.

No attempt has been made in this thesis to incorporate identities during the compilation process for the purpose of simplifying and making more efficient the resulting object program. The processes entailed in this have been adequately described in [41] and clearly merit further attention.

The research carried out in connection with this thesis has been a modest contribution to the implementation of the Iverson notation. Undoubtedly, further work on the implementation of a more complete subset of the language will be carried out. The need is certainly present.

BIBLIOGRAPHY

- [1] Adams, W.S., Probabilistic and Deterministic Aspects of Digital Computers, M.Sc. Thesis, University of Alberta, November, 1963.
- [2] Backus, J.W. et al (1959), "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Information Processings, Proceedings of ICIP at Paris, p. 125-132, UNESCO, Paris.
- [3] Baecker, H.D., "Implementing a Stack," Comm. ACM, 5, p. 505-507, (October, 1962).
- [4] Baecker, H.D., and B.J. Gibbons, "A Commercial Use of Stacks," in R. Goodman (Ed.) Annual Review in Automatic Programming, 4, Pergammon Press - MacMillan, New York, 1963.
- [5] Bailey, M.J., M.P. Barnett, and P.B. Burleson, "Symbol Manipulation in FORTRAN - SASP I Subroutines," Comm. ACM, 7, p. 339-346, (June, 1964).
- [6] Bailey, M.J., M.P. Barnett, and R.P. Futrelle, "Syntactic Analysis within Fortran - The Shadow IV System," Share Program Distribution, MIT Co-operative Computing Laboratory, 1962.
- [7] Barnett, M.P., and R.P. Futrelle, "Syntactic Analysis by Digital Computer," Comm. ACM, 5, p. 515-526, (October, 1962)
- [8] Barnett, M.P., "Low Level Language Subroutines for Use within FORTRAN," Comm. ACM, 4, p. 492-495, (Nov. 1961).
- [9] Breuer, M.A., "Techniques for the Simulation of Computer Logic," Comm. ACM, 7, p. 443-446, (July, 1964).
- [10] Brooker, R.A., "A Programming Package for Some General Modes of Arithmetic," Comm. ACM, 7, p. 119-127, (Feb. 1964).

- [11] Burks, A.W., D.W. Warren, and J.B. Wright, "An Analysis of a Logical Machine Using Parenthesis-free Notation," MTAC, 8, p. 53-57.
- [12] Burkhardt, W.H., "Metalanguage and Syntax Specification," Comm. ACM, 8, p. 304-305, (May, 1965).
- [13] Caracciola di Forino, A., "Linguistic Problems in Programming Theory," Information Processing 1965, presented at IFIP Congress, New York, May 24-29, 1965.
- [14] Caracciola di Forino, A., "Some Remarks on the Syntax of Symbolic Programming Languages," Comm. ACM, 8, p. 456-460, (Aug. 1963).
- [15] Chomsky, N., "Formal Properties of Grammers," Handbook of Mathematical Psychology, 2, 1963.
- [16] Conway, M.F., J. Speroni, "Arithmetizing Declarations: An Application to Cobol," Comm. ACM, 6, p. 24-27, (Jan. 1963).
- [17] Dijkstra, E.W., "Recursive Programming," Numerische Mathematic, 2, p. 312-318, (1960).
- [18] Eichel, J., M. Paul, F.L. Bauer, and K. Samelson, "A Syntax Controlled Generator of Formal Language Processors," Comm. ACM, 6, p. 451-455, (Aug. 1963).
- [19] Falkoff, A.D., "Algorithms for Parallel Search Memories," JACM, 9, p. 488-511, (October, 1962).
- [20] Falkoff, A.D., K.E. Iverson, E.H. Sussenguth, "A Formal Description of System/360," IBM Systems Journal, 3, p. 188-262, (1964).
- [21] Forbes, D.J., R.E. Giswold, and I.P. Polonsky, "SNOBOL, A String Manipulation Language," JACM, 11, p. 21-30, (Jan. 1964).
- [22] Floyd, R.W., "A Descriptive Language for Symbol Manipulation," JACM, 8, p. 579-584, (October, 1961).

- [23] Floyd, R.W., "Bounded Context Syntactic Analysis,"
Comm. ACM, 7, p. 62-72, (Feb. 1964).
- [24] Galler, B.A., "Compiling Matrix Operations," Comm.ACM,
5, p. 590-594, (Dec. 1962).
- [25] Gill, Stanley, "The Changing Basis of Programming,"
Information Processing 1965, Presented at IFIP
Congress, New York City, May 24-29, 1965.
- [26] Gorn, S., "An Axiomatic Approach to Prefix Languages,"
Symbolic Languages in Data Processing, Proceedings
of the Symposium of the International Computation
Centre, Rome, March 26-31, 1962. Gordon and Breech,
New York and London, 1962.
- [27] Gorn, S., Mechanical Pragmatics; "A Time-Moton Study
of a Miniature Linguistic System," Comm. ACM, 5,
p. 576-589, (Dec. 1962).
- [28] Hamblin, C.L., "Translation to and from Polish Notation,"
Computer Journal, 5, p. 210-213, (1962).
- [29] Hellerman, H., "Experimental Personalized Array
Translation System," Comm. ACM, 7, p. 433-438,
(July, 1964).
- [30] Hill, U., H. Langmaark, H.R. Schwarz, G. Seegmuller,
"Efficient Handling of Subscripted Variables in
Algol 60 Compilers," Symbolic Languages in Data
Processing, edited by the International Computational
Centre, Rome, 1962, Gordon & Breach, New York and
London, 1962.
- [31] Ingberman, P.Z., "A Translation Technique for Languages,"
Symbolic Languages in Data Processing, edited by the
International Computational Centre, Rome, 1962,
Gordon & Breach, New York and London, 1962.
- [32] Irons, E.T., "Structural Connections in Formal Languages,"
Comm. ACM, 7, p. 67-72, (Feb. 1964).

- [33] Irons, E.T., "The Structure and Use of the Syntax Directed Compiler," in R. Goodman (ed.), Annual Review in Automatic Programming, 3, Pergammon Press MacMillan, New York, 1963.
- [34] Irons, E.T., "An Error-correcting Parse Algorithm," Comm. ACM, 6, p. 669-673, (November, 1963).
- [35] Irons, E.T., "A Syntax Directed Compiler for Algol 60," Comm. ACM, 4, p. 51-55, (January, 1961).
- [36] Iverson, K.E., A Programming Language, Wiley, New York, 1962.
- [37] Iverson, K.E., "A Programming Language," Proceedings-Spring Joint Computer Conference, 1962.
- [38] Iverson, K.E., "A Common Language for Hardware, Software, and Applications," Proceedings - Fall Joint Computer Conference, 1962.
- [39] Iverson, K.E., "A Transliteration Scheme for Keying and Printing Micro Programs," IBM Research Note NC-79, Thomas J. Watson Research Center, Yorktown Heights, New York, (1962).
- [40] Iverson, K.E., "Programming Notation in Systems Design," IBM Systems Journal, 2, p. 117-128, (June, 1963).
- [41] Iverson, K.E., "Formalism in Programming Language," Comm. ACM, 7, p. 80-88, (Feb. 1954).
- [42] Iverson, K.E., "A Method of Syntax Specification," Comm. ACM, 7, p. 588-589, (October, 1964).
- [43] Iverson, K.E., Private communication, (January, 1965).
- [44] Kanner, H., P. Kosinski, and C.L. Robinson, "The Structure of Yet Another ALGOL Compiler," Comm. ACM, 8, p. 427-438, (July, 1965).
- [45] Knuth, Donald E., "Bachus Normal Form vs Bachus Naur Form," Comm. ACM, 7, p. 735-736, (Dec. 1964).

- [46] Larson, R.P., M.M. Mailo, " Modeling and Simulation of Digital Networks," Comm. ACM, 8, p. 308-312, (May, 1965).
- [47] Leavenworth, B.M., "Fortran IV as a Syntax Language," Comm. ACM, 7, p. 72-80, (Feb. 1964).
- [48] Leavenworth, B.M., Private communication, (March, 1965).
- [49] Ledley, R.S., and B. Wilson, "Automatic-Programming Language Translation Through Syntactical Analysis," Comm. ACM, 5, p. 145-155, (March, 1962).
- [50] Lukasiewicz, J., Aristotles Syllogistic from the Standpoint of Modern Formal Logic, Clarendon Press, Oxford, England, p. 78.
- [51] Meggitt, J.E., "A Character Computer for a High-level Language," IBM Systems Journal, 3, p. 68-78, (1964).
- [52] Morris, D., "Syntactical Analysis in Compilers, Chapter 19 of P. Wegner (Ed.)," Introduction to System Programming, Academic Press, London and New York, (1964).
- [53] Naur, Peter (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, 6, (Jan. 1963). p. 1-17.
- [54] Oettinger, A.G., "Automatic Syntactic Analysis and the Pushdown Store," Proceedings of the Twelfth Symposium in Applied Mathematics, p. 104-129, published by the Americal Mathematical Society, Providence, R.I., 1961.
- [55] Petrich, S.R., " More on Bachus Normal Form," Comm. ACM, 8, p. 200, (March, 1965).
- [56] Randell, B., "The Whetstone KDF9 Algol Translator, Chapter 8 of P. Wegner (Ed.)," Introduction to System Programming, Academic Press, London and New York, (1964).

- [57] Randell, B., and L.J. Russell, Algol 60 Implementation, Academic Press, London and New York, 1964, p. 8-33, p. 149-171.
- [58] Randell, B., and L.J., Russell, "Single Scan Techniques for the Translation of Arithmetic Expressions in Algol 60," JACM, 11, p. 159-167, (April, 1964).
- [59] Rosen, S., "A Compiler-Building System Developed by Brooker and Morris," Comm. ACM, 7, p. 403-414, (July, 1964).
- [60] Salton, G., "Manipulation of Trees in Information Retrieval," Comm. ACM, 5, p. 103-114, (Feb. 1962).
- [61] Sensig, D.M., "Suggested Timing Notation for the Iverson Notation," IBM Research Note NC-120, Thomas J. Watson Research Center, Yorktown Heights, N.Y., (1962).
- [62] Wegner, P., "Stack Compilation Techniques," Chapter 7, of P. Wegner (Ed.), Introduction to System Programming, Academic Press, London and New York, (1964).
- [63] Wegstein, J.H., and W.W. Youden, "A String Language for Symbol Manipulation Based on Algol 60," Comm. ACM, 9, p. 54-61, (Jan. 1962).

APPENDIX A

Implemented Subset of Iverson's Language

The subset of Iverson's Language which is recognizable to the compiler written for the IBM 7040 is approximately that subset used in Chapter II of [36], Chapter V of [1] and [20]. That is, the operators and relations are those which would be used in machine logic design in the manner suggested in the above references. The selection operators form the heart of this subset. A number of other manipulative operators such as catenation and rotation are also included. Most logical operators and a sufficient number of integer operators for performing array indexing and machine arithmetic are also recognizable. The allowable operands are thus restricted to being logical or integer scalars, vectors and matrices. Set operations, trees, files, mappings and permutations are omitted completely.

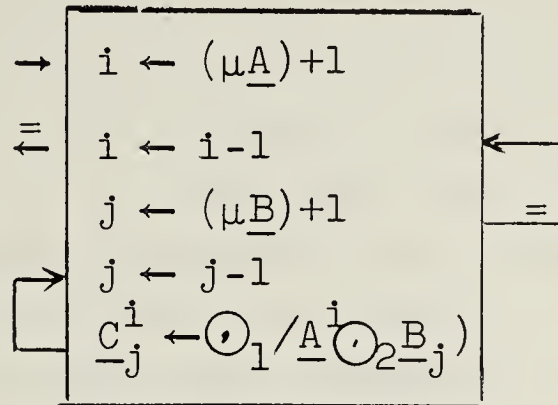
Vectors and matrices representing machine registers or memory units do not vary in dimension throughout the runnings of a program. Hence, none of the dimension operators (e.g. μ and ν) are needed for this subset. Even a stack, which is conveniently described as being of variable dimension (see program 2.1) will be physically represented by a vector of constant dimension.

Table B.1 shows the set of operators which are recognizable. A few other relations and arithmetic operators could be added merely by enlarging some tables.

The special vectors $\underline{\omega}^i(n)$, $\underline{\alpha}^i(n)$, $\underline{\lambda}^i(n)$ and $\underline{\epsilon}^i(n)$ are also recognizable, but none of the special matrices are presently included. Once again, the modification of a few tables and the choice of a suitable transliteration scheme would allow these to be included. The generalized matrix product has been omitted, mainly because of the lack of a suitable transliteration.

For vectors, the equivalent to $\underline{a} \circledcirc \underline{b}$ is readily re-written as $\circledcirc_1 / (\underline{a} \circledcirc_2 \underline{b})$. The extension to matrices is not neatly rewritten in terms of simpler operations.

Thus, $\underline{c} \leftarrow \underline{A} \circledcirc \underline{B}$ must be written as:



APPENDIX B

Transliteration Scheme

The following describes the transliteration scheme which was used in the implementation of a "machine design subset" of Iverson's language on the IBM 7040. Although it would be desirable to use all sixty-four characters in the IBM 7040 character set, the fact that only forty-eight (10 numeric, 26 alphabetic and 12 special) characters are available on the standard 1403 print chain precluded this. Consequently, operators and relations were transliterated to mnemonic multiletter codes. These codes are enclosed between periods, as in FORTRAN IV and MAD. Thus $\wedge$ becomes .AND., $\uparrow$ becomes .LRT. etc. This is shown fully in table B.1. The alternate notation for tertiary operations presented Chapter III is used. In place of a semi-colon, an apostrophe is used as a delimiter. Thus $/\underline{a}, \underline{\mu}, \underline{b}/$ is written $U()/A()'B()$.

Indices for structured operands are written to the right of the identifier and are enclosed in parenthesis and separated by a dollar sign, \$. Null indices are denoted by the procedure of omitting them completely. Thus $\underline{a}_i$ is written $A(I)$, $\underline{M}_j$ is written $M(\$J)$ and $\underline{U}$ is written $U(\$)$ to denote the entire matrix. The row index (printed as a superscript) precedes the column index, (printed as a subscript) under this transliteration scheme.

Table B.1

OPERATION	PRINTED NOTATION	TRANSLITERATED NOTATION
Sum	$a+b$	$A+B$
Difference	$a-b$ or $-a$	$A-B$ or $-A$
Product	$a \times b$	$A*B$
Negation	$\bar{\mu}$ or $\sim\mu$	$.NOT. U$
And	$\mu \wedge \nu$	$U .AND. V$
Or	$\mu \vee \nu$	$U .OR. V$
Residue modulo m	$m \mid n$	$M .MOD. N$
Equality relation	$a=b$	$A.EQ.B$
Inequality relation	$a \neq b$	$A.NE.B$
Base 2 value	$\perp \mu$	$.BASE. U()$
Representation base 2	$(n) \top j$	$N.REP. J$
Catenation	$\underline{x}, \underline{y}$	$X(), Y()$
Compression (row)	$\underline{\mu}/\underline{a}$	$U()/A()$
(column)	$\underline{\mu}/\underline{\underline{X}}$	$U()/\underline{\underline{X}}()$
Expansion (row)	$\underline{\mu} \backslash \underline{a}$	$U() \nmid A()$
(column)	$\underline{\mu} \backslash \underline{\underline{a}}$	$U() \nmid \underline{\underline{A}}()$
Mask (row)	$\underline{\underline{a}}; \underline{\mu}; \underline{b} /$ or $\underline{\mu}/\underline{a}; \underline{b}$	$U()/A() \mid B()$
Mesh (row)	$\backslash \underline{a}; \underline{\mu}; \underline{b} /$ or $\underline{\mu} \backslash \underline{a}; \underline{b}$	$U() \nmid A() \mid B()$
Mask (column)	$// \underline{\underline{A}}; \underline{\mu}; \underline{\underline{B}} //$ or $\underline{\mu} // \underline{\underline{A}}; \underline{\underline{B}}$	$U() // A() \mid B()$
Mesh (column)	$\underline{\underline{A}}; \underline{\mu}; \underline{\underline{B}}$ or $\underline{\mu} \underline{\underline{A}}; \underline{\underline{B}}$	$U() \nmid \underline{\underline{A}}() \mid B()$
Left Rotation	$k \uparrow \underline{a}$	$K.LRT.A()$
Right Rotation	$k \downarrow \underline{a}$	$K.RRT.B()$
Reduction	$\odot /$ where $\bigcirc$ is one of: $= \neq \wedge \vee + \times - ,$	$\odot /$ where $\bigcirc$ is one of: $.EQ. .NE.$ $.AND. .OR. + * - ,$

APPENDIX C

Description of 7040 Iverson Compiler

The Iverson compiler which was written for the IBM 7040 is divided at present into 15 separate decks. One of these is written in COBOL, six are written in FORTRAN and the remaining eight are written in MAP. The language is indicated in brackets after the name of each routine. Each of these decks will be described in the following:

1. EDIT [COBOL]

This deck performs an initial edit of the source program. The major accomplishment of this edit is to change the program from the transliterated source to the internal code described in APPENDIX D. Thus, only this program need be modified, to alter the transliteration scheme. The source listing is produced by this program.

Detection of unary plus and minus occurs here. Unary minus is replaced by a code for .NEG. while unary plus is omitted from the internal source string.

The existence of a null index is detected here also. It is indicated as being explicitly null in the resulting internal code.

This program controls the entire compilation process and as a consequence, calls the subroutines ISCAN and GENCOD. Several others routines are used by this program, principally, those contained in deck TABLE, which handles the symbol table.

2. HEADG [MAP]

This deck performs the simple task of initializing the SUBHD field in the output editor to the heading desired for the listing of the Iverson source statements. It is called by EDIT.

3. RANDAD [MAP]

This routine takes the double word alphameric-symbol for a transliterated operation code and converts it to a number between 1 and 64. This number is used by EDIT as the entry point for a table search. The two halves of the double word symbol are added together while still in BCD code and then the result is added, character by character with the high order bits truncated (i.e., only the low order 6 bits are retained). The result is this entry index.

4. CHARCD [MAP]

This deck is used by EDIT to provide a quick method of classifying input characters. It provides a code number between 1 and 11 which classifies the character as to numeric, alphabetic or one of several separate classifications of special characters.

5. TABLE [MAP]

This routine manipulates the symbol table. It is used by decks EDIT, ISCAN, GENCOD and the deck which handles declaratives (at present TMPINT). It has several entry points, one for each possible function associated with a symbol table. The single word containing the identifier in BCD form is compressed to 9 bits by "folding" twice to give a number between 0 and 511. This is used as the entry index for the table search. The maximum number of allowable symbols is at present 512. This could be increased readily by modifying the search routines in this deck.

6. EDCODS [MAP]

This deck consists entirely of 3 tables. First, the table of transliterated operations each in a two word BCD entry. Second, a table of equivalent internal codes. Third, the entry point in the runtime subroutines which correspond to those operations. The first table is used by EDIT, the second by EDIT and ISCAN and the third by GENCOD.

BIBLIOGRAPHY

- [1] Adams, W.S., Probabilistic and Deterministic Aspects of Digital Computers, M.Sc. Thesis, University of Alberta, November, 1963.
- [2] Backus, J.W. et al (1959), "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," Information Processings, Proceedings of ICIP at Paris, p. 125-132, UNESCO, Paris.
- [3] Baecker, H.D., "Implementing a Stack," Comm. ACM, 5, p. 505-507, (October, 1962).
- [4] Baecker, H.D., and B.J. Gibbons, "A Commercial Use of Stacks," in R. Goodman (Ed.) Annual Review in Automatic Programming, 4, Pergammon Press - MacMillan, New York, 1963.
- [5] Bailey, M.J., M.P. Barnett, and P.B. Burleson, "Symbol Manipulation in FORTRAN - SASP I Subroutines," Comm. ACM, 7, p. 339-346, (June, 1964).
- [6] Bailey, M.J., M.P. Barnett, and R.P. Futrelle, "Syntactic Analysis within Fortran - The Shadow IV System," Share Program Distribution, MIT Co-operative Computing Laboratory, 1962.
- [7] Barnett, M.P., and R.P. Futrelle, "Syntactic Analysis by Digital Computer," Comm. ACM, 5, p. 515-526, (October, 1962)
- [8] Barnett, M.P., "Low Level Language Subroutines for Use within FORTRAN," Comm. ACM, 4, p. 492-495, (Nov. 1961).
- [9] Breuer, M.A., "Techniques for the Simulation of Computer Logic," Comm. ACM, 7, p. 443-446, (July, 1964).
- [10] Brooker, R.A., "A Programming Package for Some General Modes of Arithmetic," Comm. ACM, 7, p. 119-127, (Feb. 1964).

- [11] Burks, A.W., D.W. Warren, and J.B. Wright, "An Analysis of a Logical Machine Using Parenthesis-free Notation," MTAC, 8, p. 53-57.
- [12] Burkhardt, W.H., "Metalanguage and Syntax Specification," Comm. ACM, 8, p. 304-305, (May, 1965).
- [13] Caracciola di Forino, A., "Linguistic Problems in Programming Theory," Information Processing 1965, presented at IFIP Congress, New York, May 24-29, 1965.
- [14] Caracciola di Forino, A., "Some Remarks on the Syntax of Symbolic Programming Languages," Comm. ACM, 8, p. 456-460, (Aug. 1963).
- [15] Chomsky, N., "Formal Properties of Grammars," Handbook of Mathematical Psychology, 2, 1963.
- [16] Conway, M.F., J. Speroni, "Arithmetizing Declarations: An Application to Cobol," Comm. ACM, 6, p. 24-27, (Jan. 1963).
- [17] Dijkstra, E.W., "Recursive Programming," Numerische Mathematic, 2, p. 312-318, (1960).
- [18] Eichel, J., M. Paul, F.L. Bauer, and K. Samelson, "A Syntax Controlled Generator of Formal Language Processors," Comm. ACM, 6, p. 451-455, (Aug. 1963).
- [19] Falkoff, A.D., "Algorithms for Parallel Search Memories," JACM, 9, p. 488-511, (October, 1962).
- [20] Falkoff, A.D., K.E. Iverson, E.H. Sussenguth, "A Formal Description of System/360," IBM Systems Journal, 3, p. 188-262, (1964).
- [21] Forbes, D.J., R.E. Giswold, and I.P. Polonsky, "SNOBOL, A String Manipulation Language," JACM, 11, p. 21-30, (Jan. 1964).
- [22] Floyd, R.W., "A Descriptive Language for Symbol Manipulation," JACM, 8, p. 579-584, (October, 1961).

- [23] Floyd, R.W., "Bounded Context Syntactic Analysis," Comm. ACM, 7, p. 62-72, (Feb. 1964).
- [24] Galler, B.A., "Compiling Matrix Operations," Comm.ACM, 5, p. 590-594, (Dec. 1962).
- [25] Gill, Stanley, "The Changing Basis of Programming," Information Processing 1965, Presented at IFIP Congress, New York City, May 24-29, 1965.
- [26] Gorn, S., "An Axiomatic Approach to Prefix Languages," Symbolic Languages in Data Processing, Proceedings of the Symposium of the International Computation Centre, Rome, March 26-31, 1962. Gordon and Breech, New York and London, 1962.
- [27] Gorn, S., Mechanical Pragmatics; "A Time-Moton Study of a Miniature Linguistic System," Comm. ACM, 5, p. 576-589, (Dec. 1962).
- [28] Hamblin, C.L., "Translation to and from Polish Notation," Computer Journal, 5, p. 210-213, (1962).
- [29] Hellerman, H., "Experimental Personalized Array Translation System," Comm. ACM, 7, p. 433-438, (July, 1964).
- [30] Hill, U., H. Langmaark, H.R. Schwarz, G. Seegmuller, "Efficient Handling of Subscripted Variables in Algol 60 Compilers," Symbolic Languages in Data Processing, edited by the International Computational Centre, Rome, 1962, Gordon & Breach, New York and London, 1962.
- [31] Ingerman, P.Z., "A Translation Technique for Languages," Symbolic Languages in Data Processing, edited by the International Computational Centre, Rome, 1962, Gordon & Breach, New York and London, 1962.
- [32] Irons, E.T., "Structural Connections in Formal Languages," Comm. ACM, 7, p. 67-72, (Feb. 1964).

- [33] Irons, E.T., "The Structure and Use of the Syntax Directed Compiler," in R. Goodman (ed.), Annual Review in Automatic Programming, 3, Pergammon Press MacMillan, New York, 1963.
- [34] Irons, E.T., "An Error-correcting Parse Algorithm," Comm. ACM, 6, p. 669-673, (November, 1963).
- [35] Irons, E.T., "A Syntax Directed Compiler for Algol 60," Comm. ACM, 4, p. 51-55, (January, 1961).
- [36] Iverson, K.E., A Programming Language, Wiley, New York, 1962.
- [37] Iverson, K.E., "A Programming Language," Proceedings-Spring Joint Computer Conference, 1962.
- [38] Iverson, K.E., "A Common Language for Hardware, Software, and Applications," Proceedings - Fall Joint Computer Conference, 1962.
- [39] Iverson, K.E., "A Transliteration Scheme for Keying and Printing Micro Programs," IBM Research Note NC-79, Thomas J. Watson Research Center, Yorktown Heights, New York, (1962).
- [40] Iverson, K.E., "Programming Notation in Systems Design," IBM Systems Journal, 2, p. 117-128, (June, 1963).
- [41] Iverson, K.E., "Formalism in Programming Language," Comm. ACM, 7, p. 80-88, (Feb. 1954).
- [42] Iverson, K.E., "A Method of Syntax Specification," Comm. ACM, 7, p. 588-589, (October, 1964).
- [43] Iverson, K.E., Private communication, (January, 1965).
- [44] Kanner, H., P. Kosinski, and C.L. Robinson, "The Structure of Yet Another ALGOL Compiler," Comm. ACM, 8, p. 427-438, (July, 1965).
- [45] Knuth, Donald E., "Bachus Normal Form vs Bachus Naur Form," Comm. ACM, 7, p. 735-736, (Dec. 1964).

- [46] Larson, R.P., M.M. Mailo, " Modeling and Simulation of Digital Networks," Comm. ACM, 8, p. 308-312, (May, 1965).
- [47] Leavenworth, B.M., "Fortran IV as a Syntax Language," Comm. ACM, 7, p. 72-80, (Feb. 1964).
- [48] Leavenworth, B.M., Private communication, (March, 1965).
- [49] Ledley, R.S., and B. Wilson, "Automatic-Programming Language Translation Through Syntactical Analysis," Comm. ACM, 5, p. 145-155, (March, 1962).
- [50] Lukasiewicz, J., Aristotles Syllogistic from the Standpoint of Modern Formal Logic, Clarendon Press, Oxford, England, p. 78.
- [51] Meggitt, J.E., "A Character Computer for a High-level Language," IBM Systems Journal, 3, p. 68-78, (1964).
- [52] Morris, D., "Syntactical Analysis in Compilers, Chapter 19 of P. Wegner (Ed.)," Introduction to System Programming, Academic Press, London and New York, (1964).
- [53] Naur, Peter (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, 6, (Jan. 1963). p. 1-17.
- [54] Oettinger, A.G., "Automatic Syntactic Analysis and the Pushdown Store," Proceedings of the Twelfth Symposium in Applied Mathematics, p. 104-129, published by the Americal Mathematical Society, Providence, R.I., 1961.
- [55] Petrich, S.R., " More on Bachus Normal Form," Comm. ACM, 8, p. 200, (March, 1965).
- [56] Randell, B., "The Whetstone KDF9 Algol Translator, Chapter 8 of P. Wegner (Ed.)," Introduction to System Programming, Academic Press, London and New York, (1964).

- [57] Randell, B., and L.J. Russell, Algol 60 Implementation, Academic Press, London and New York, 1964, p. 8-33, p. 149-171.
- [58] Randell, B., and L.J., Russell, "Single Scan Techniques for the Translation of Arithmetic Expressions in Algol 60," JACM, 11, p. 159-167, (April, 1964).
- [59] Rosen, S., "A Compiler-Building System Developed by Brooker and Morris," Comm. ACM, 7, p. 403-414, (July, 1964).
- [60] Salton, G., "Manipulation of Trees in Information Retrieval," Comm. ACM, 5, p. 103-114, (Feb. 1962).
- [61] Sensig, D.M., "Suggested Timing Notation for the Iverson Notation," IBM Research Note NC-120, Thomas J. Watson Research Center, Yorktown Heights, N.Y., (1962).
- [62] Wegner, P., "Stack Compilation Techniques," Chapter 7, of P. Wegner (Ed.), Introduction to System Programming, Academic Press, London and New York, (1964).
- [63] Wegstein, J.H., and W.W. Youden, "A String Language for Symbol Manipulation Based on Algol 60," Comm. ACM, 9, p. 54-61, (Jan. 1962).

APPENDIX A

Implemented Subset of Iverson's Language

The subset of Iverson's Language which is recognizable to the compiler written for the IBM 7040 is approximately that subset used in Chapter II of [36], Chapter V of [1] and [20]. That is, the operators and relations are those which would be used in machine logic design in the manner suggested in the above references. The selection operators form the heart of this subset. A number of other manipulative operators such as catenation and rotation are also included. Most logical operators and a sufficient number of integer operators for performing array indexing and machine arithmetic are also recognizable. The allowable operands are thus restricted to being logical or integer scalars, vectors and matrices. Set operations, trees, files, mappings and permutations are omitted completely.

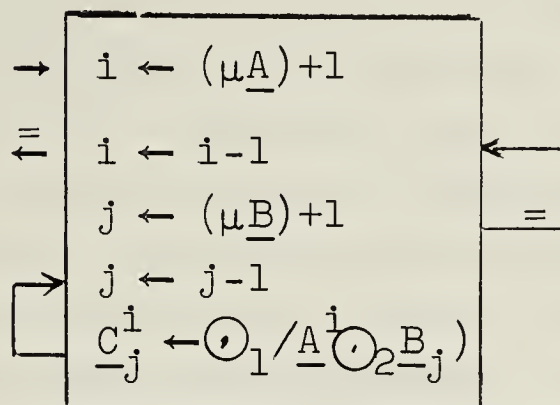
Vectors and matrices representing machine registers or memory units do not vary in dimension throughout the runnings of a program. Hence, none of the dimension operators (e.g. μ and ν) are needed for this subset. Even a stack, which is conveniently described as being of variable dimension (see program 2.1) will be physically represented by a vector of constant dimension.

Table B.1 shows the set of operators which are recognizable. A few other relations and arithmetic operators could be added merely by enlarging some tables.

The special vectors $\underline{\omega}^i(n)$, $\underline{\alpha}^i(n)$, $\underline{1}^i(n)$ and $\underline{\epsilon}^i(n)$ are also recognizable, but none of the special matrices are presently included. Once again, the modification of a few tables and the choice of a suitable transliteration scheme would allow these to be included. The generalized matrix product has been omitted, mainly because of the lack of a suitable transliteration.

For vectors, the equivalent to $\underline{a} \circledast \underline{b}$ is readily re-written as $\odot_1 / (\underline{a} \odot_2 \underline{b})$. The extension to matrices is not neatly rewritten in terms of simpler operations.

Thus, $\underline{c} \leftarrow \underline{A} \circledast \underline{B}$ must be written as:



APPENDIX B

Transliteration Scheme

The following describes the transliteration scheme which was used in the implementation of a "machine design subset" of Iverson's language on the IBM 7040. Although it would be desirable to use all sixty-four characters in the IBM 7040 character set, the fact that only forty-eight (10 numeric, 26 alphabetic and 12 special) characters are available on the standard 1403 print chain precluded this. Consequently, operators and relations were transliterated to mnemonic multiletter codes. These codes are enclosed between periods, as in FORTRAN IV and MAD. Thus $\wedge$ becomes .AND., $\uparrow$ becomes .LRT. etc. This is shown fully in table B.1. The alternate notation for tertiary operations presented Chapter III is used. In place of a semi-colon, an apostrophe is used as a delimiter. Thus $/\underline{a}, \underline{u}, \underline{b}/$ is written $U()/A()'B()$.

Indices for structured operands are written to the right of the identifier and are enclosed in parenthesis and separated by a dollar sign, \$. Null indices are denoted by the procedure of omitting them completely. Thus $\underline{a}_i$ is written $A(I)$, $\underline{M}_j$ is written $M(\$J)$ and $\underline{U}$ is written $U(\$)$ to denote the entire matrix. The row index (printed as a superscript) precedes the column index, (printed as a subscript) under this transliteration scheme.

Table B.1

OPERATION	PRINTED NOTATION	TRANSLITERATED NOTATION
Sum	$a+b$	$A+B$
Difference	$a-b$ or $-a$	$A-B$ or $-A$
Product	$a \times b$	$A*B$
Negation	$\bar{\mu}$ or $\sim\mu$	$.NOT. U$
And	$\mu \wedge \nu$	$U .AND. V$
Or	$\mu \vee \nu$	$U .OR. V$
Residue modulo m	$m \mid n$	$M .MOD. N$
Equality relation	$a=b$	$A.EQ.B$
Inequality relation	$a \neq b$	$A.NE.B$
Base 2 value	$\perp \mu$	$.BASE. U()$
Representation base 2	$(n) \top j$	$N.REP. J$
Catenation	$\underline{x}, \underline{y}$	$X(), Y()$
Compression (row)	$\underline{\mu}/\underline{a}$	$U()/A()$
(column)	$\underline{\mu} // \underline{X}$	$U()//X()$
Expansion (row)	$\underline{\mu} \backslash \underline{a}$	$U() \nmid A()$
(column)	$\underline{\mu} \backslash \backslash \underline{a}$	$U() \nmid \nmid A()$
Mask (row)	$\underline{a}; \underline{\mu}; \underline{b} /$ or $\underline{\mu}/\underline{a}; \underline{b}$	$U()/A() \nmid B()$
Mesh (row)	$\backslash \underline{a}; \underline{\mu}; \underline{b} \backslash$ or $\underline{\mu} \backslash \underline{a}; \underline{b}$	$U() \nmid A() \nmid B()$
Mask (column)	$// \underline{A}; \underline{\mu}; \underline{B} //$ or $\underline{\mu} // \underline{A}; \underline{B}$	$U()//A() \nmid B()$
Mesh (column)	$\underline{A}; \underline{\mu}; \underline{B}$ or $\underline{\mu} \underline{A}; \underline{B}$	$U() \nmid \nmid A() \nmid B()$
Left Rotation	$k \uparrow \underline{a}$	$K.LRT.A()$
Right Rotation	$k \downarrow \underline{a}$	$K.RRT.B()$
Reduction	$\odot /$ where $\odot$ is one of: $= \neq \wedge \vee + \times -$,	$\odot /$ where $\odot$ is one of: $.EQ. .NE. .AND. .OR. + * -$

APPENDIX C

Description of 7040 Iverson Compiler

The Iverson compiler which was written for the IBM 7040 is divided at present into 15 separate decks. One of these is written in COBOL, six are written in FORTRAN and the remaining eight are written in MAP. The language is indicated in brackets after the name of each routine. Each of these decks will be described in the following:

1. EDIT [COBOL]

This deck performs an initial edit of the source program. The major accomplishment of this edit is to change the program from the transliterated source to the internal code described in APPENDIX D. Thus, only this program need be modified, to alter the transliteration scheme. The source listing is produced by this program.

Detection of unary plus and minus occurs here. Unary minus is replaced by a code for .NEG. while unary plus is omitted from the internal source string.

The existence of a null index is detected here also. It is indicated as being explicitly null in the resulting internal code.

This program controls the entire compilation process and as a consequence, calls the subroutines ISCAN and GENCOD. Several others routines are used by this program, principally, those contained in deck TABLE, which handles the symbol table.

2. HEADG [MAP]

This deck performs the simple task of initializing the SUBHD field in the output editor to the heading desired for the listing of the Iverson source statements. It is called by EDIT.

3. RANDAD [MAP]

This routine takes the double word alphameric-symbol for a transliterated operation code and converts it to a number between 1 and 64. This number is used by EDIT as the entry point for a table search. The two halves of the double word symbol are added together while still in BCD code and then the result is added, character by character with the high order bits truncated (i.e., only the low order 6 bits are retained). The result is this entry index.

4. CHARCD [MAP]

This deck is used by EDIT to provide a quick method of classifying input characters. It provides a code number between 1 and 11 which classifies the character as to numeric, alphabetic or one of several separate classifications of special characters.

5. TABLE [MAP]

This routine manipulates the symbol table. It is used by decks EDIT, ISCAN, GENCOD and the deck which handles declaratives (at present TMPINT). It has several entry points, one for each possible function associated with a symbol table. The single word containing the identifier in BCD form is compressed to 9 bits by "folding" twice to give a number between 0 and 511. This is used as the entry index for the table search. The maximum number of allowable symbols is at present 512. This could be increased readily by modifying the search routines in this deck.

6. EDCODS [MAP]

This deck consists entirely of 3 tables. First, the table of transliterated operations each in a two word BCD entry. Second, a table of equivalent internal codes. Third, the entry point in the runtime subroutines which correspond to those operations. The first table is used by EDIT, the second by EDIT and ISCAN and the third by GENCOD.

7. ISCAN [FORTRAN]

This deck is written in FORTRAN and performs the translation to a Lukasiewicz string from the Iverson source statement, which is in an internal code. The methods used are those described in Chapter III. Function calls and indexed variables are detected here. This subroutine is called from EDIT, once for each source statement.

8. GENCOD [FORTRAN]

This deck generates the MAP coding from the Lukasiewicz string. All statements produced are call statements, but macros could be produced just as easily, with only minor modification. The algorithm used is that described in program 2.2. This subroutine is called from EDIT immediately following the call to ISCAN.

9. STACK [MAP]

This routine simulates a stack. It has several entry points, one each possible stack manipulation (clearing the stack, stacking an item, unstacking an item, examining the top item in the stack, etc.). It is called by both ISCAN and GENCOD. At present, up to 500 items may be in the stack at any one time.

10. PREC [FORTRAN]

This routine returns a precedence value for the single operator which is its argument. The operator is in the internal code. ISCAN is the only deck which refers to this function.

11. UQBC [MAP]

This routine returns the BCD representation of the function argument (a binary number). A positive binary integer up to six decimal digits in length will be converted. Any number outside this range will return the value 999999. It is used by GENCOD in generating names for temporary storage.

12. CODFIL [MAP]

This is used to open and close the file which contains the compiled input for IBCMAP. This is presently the FTCD4 file. A "\$IBMAP" card image is generated upon opening and a "END" card image is generated just prior to closing. In addition, a counter used by GENCOD is reset upon opening. Both entry points are called from EDIT.

13. ERRGEN [FORTRAN]

This routine generates on the output file, the error messages corresponding to the error codes produced by all other routines during compilation. It is called from EDIT just prior to closing all files and terminating compilation.

14. TMPINT [FORTRAN]

This is a routine which provides a temporary method of entering symbols into the symbol table. It should be replaced by a routine which recognizes a declaration in the input string. All that is necessary to enter a symbol into the symbol table is to call ENTRTB in deck TABLE. Three arguments are necessary for this call: the BCD representation of the symbol, the IVSN number (produced in deck EDIT) and the type code for the particular symbol (see APPENDIX D).

15. TMPDEC [FORTRAN]

This routine provides a temporary method of placing declaratives in the file of MAP coding (FTCD4). It should be replaced by a routine which searches the symbol table and outputs declaratives for all symbols. Also necessary are EXTERNS for all runtime subroutine names. These should be under the UNLIST option. At the completion of compilation, MAXT in the control section BCD contains the maximum value reached by the counter of temporary storage areas. This would be used to produce declaratives for "R.xxxx" and "T.xxxx" symbols.

Any prospective user of the above routines should be aware of the design philosophy of the above routines. That is, they are intended as a pilot model to develop the compiling techniques for Iverson's Language. They are not intended to be used in a production type compiler while in their present form. The fact that the permanent routines for declaratives are not included is further illustration of the limitations of the compiler while in its present form. On the other hand, many of the routines, for example, the routines for the symbol table, are programmed in an efficient manner.

APPENDIX D

Description of the Internal Code

As noted in APPENDIX C, the routine EDIT translates the transliterated Iverson language string into an internal code for use by the programs ISCAN and GENCOD. This code consists of series of integers, stored one per word with each integer corresponding to a single source entity. All identifiers are coded as a negative numbers and all delimiters and operators as positive numbers. Right and left parenthesis are coded as 1 and 2 respectively. The replacement operator is given the value 3. The delimiter for function calls and for indices is coded as a 4. The delimiter used in the alternate form for tertiary selection operators described in CHAPTER III is recoded as a 5. All unary operators including various reduction operations are given unique codes between 6 and 19 inclusive. All remaining binary and tertiary operators are given semi-unique codes between 20 and 38 inclusive. The routine ISCAN makes these unique by differentiating between the binary and tertiary correspondents some operators. The tertiary ones are changed to a range 45 to 48 inclusive.

The function call and indexing operators must have a degree associated with them when in the Lukasiewicz string. (This is no problem for other operators since it is implied for them). This is done by storing the degree in the decrement field (bits 3-17 incl) of the word which contains the integer code. ISCAN performs this task.

All identifiers are given a code between -1 and -512. The absolute value of this code is the same as the entry index used in TABLE which is produced by "folding" the BCD representation of this identifier. Special vectors are coded between -2000 and -2003 inclusive. Any literal are coded between -2101 and -2999 inclusive. The null subscript is given the code of -3000.

In addition to these, there is a type code associated with each variable. It is also an integer and has the value 1 for a function call, 2 for a scalar, 3 for a vector and 4 for a matrix. This code is assigned during handling of declarative and is contained in a companion table to the symbol table.

APPENDIX E

The Grid Matrix on the 7040

On the 7040, the modified grid matrix scheme referred to in section 4.5 is to be used for the representation of vectors or other structured operands stored as vectors. These vectors are stored piecewise in a packed format. i.e., each vector is split into a sequence of consecutive elements. (each referred to as an infix). Each infix is packed into consecutive words in storage with integer vectors packed 1 element for word and logical vectors packed 32 elements per word.

In all cases, only the left hand 32 bits are used. The location of these infixes is described by a grid matrix for each vector. For each infix, there corresponds one entry in the grid matrix which is stored in a modified chained format, one entry per word. Infixes of the grid matrix are stored in a linear fashion. The end of each infix is demarked by the presence of a "pointer" word which has a one bit in the sign position and the address of the next infix in positions 21-35. If this address is zero, then it is also the end of the vector.

The format of each grid matrix entry is as follows:

bit s	- pointer indicator
bit 1-3	- binary representation of $\log_2$ of number of bits occupied by each element (e.g., 001) for logical and 101 for integer.
bits 4-23	- element address. For an integer, only bits 4-18 are used and this is the word address. For a logical vector, bits 4-18 give the word address and bits 18-23 give the bit address.
24 - 35	- number of elements in infix.

Thus, for an integer vector of 10 elements stored in two infixes, the first of length 3, starting at address 00100₈

and the second of length 7, starting at address 00177₈, the grid matrix starting at address 00050₈ would consist of (in octal);

00050	+	5	00100	00	0003
00051	+	5	00177	00	0007
00052	-		0000000		00000

If the second entry were stored at address 00060, then the grid matrix would consist of:

00050	+	5	00100	00	0003
00051	-		0000000		00060
00060	+	5	00177	00	0007
00061	-		00000	00	00000

If the vector was logical vector, the first infix starting at bit 0 of word 00100, and the second infix starting at bit 21₈ of word 0177₈, then the grid matrix would be:

00050	+	1	00100	00	0003
00051	+	1	00177	21	0007
00052	-		0000000		00000

This scheme is easily applied to operands of other lengths so long as the size (in bits) is a power of 2 and is less than 2⁷=128 bits. Furthermore, elements of a vector may be of varying lengths, provided each infix is of a constant size. The interpretation to be used when for example, to operands of different lengths are added (e.g., does the shorter one have a sign bit?) has not been investigated fully.

APPENDIX F

Diagram F.1 shows a series of statements, both in their transliterated form and in the internal code. The first set of codes show the statements as produced by EDIT. The second set of codes show them in the Lukasiewicz string produced by ISCAN.

Diagram F.2 shows the same statements in the MAP code produced by GENCOD and assembled by IBMAP.

IVERSON SOURCE STATEMENTS

MAR()=MARS(0\$)

2

```
MBR( )=M(+ /BETA*(.NOT..W(1$24)/MAR( )$))
```

3

```
OR()=(MDR(0).NE.MDR(1)),(.W(4$6)/MDR()),(MBR(6).NE.MBR(7)),(.W(4$1  
12)/MBR())
```

4

$$MDR() = MAR(23) / (.A(6\$12) / MBR()) * (.W(6\$12) / MBR())$$

5

END

Table F.2

	CALL	I.SUBS(R.0001,T.0001,MARS...=00004,L.0001,=00002,=00000,	29
	ETC	=00000)	
	CALL	I.SUBS(R.0002,T.0002,MAR...=00003,=00000,=00000)	30
BINARY CARD 00002			
	CALL	I.SPEC(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	31
	CALL	I.SUBS(R.0001,T.0001,MAR...=00003,=00000,=00000)	32
BINARY CARD 00003			
	CALL	OMEGA(R.0002,T.0002,L.0002,=00002,L.0003,=00002)	33
	CALL	I.CMPR(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	34
BINARY CARD 00004			
	CALL	I.NDT.(R.0001,T.0001,R.0001,T.0001)	35
	CALL	I.MULT(R.0001,T.0001,BETA...=00003,R.0001,T.0001)	36
BINARY CARD 00005			
	CALL	I.PLRD(R.0001,T.0001,R.0001,T.0001)	37
	CALL	I.SUBS(R.0001,T.0001,M...=00004,R.0001,T.0001,=00000,	38
	ETC	=00000)	
	CALL	I.SUBS(R.0002,T.0002,MBR...=00003,=00000,=00000)	39
BINARY CARD 00006			
	CALL	I.SPEC(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	40
	CALL	I.SUBS(R.0001,T.0001,MBR...=00003,=00000,=00000)	41
BINARY CARD 00007			
	CALL	OMEGA(R.0002,T.0002,L.0004,=00002,L.0007,=00002)	42
	CALL	I.CMPR(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	43
BINARY CARD 00010			
	CALL	I.SUBS(R.0002,T.0002,MBR...=00003,L.0006,=00002)	44
	CALL	I.SUBS(R.0003,T.0003,MBR...=00003,L.0005,=00002)	45
BINARY CARD 00011			
	CALL	I.NE..(R.0002,T.0002,R.0003,T.0003,R.0002,T.0002)	46
	CALL	I.CAT.(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	47
BINARY CARD 00012			
	CALL	I.SUBS(R.0002,T.0002,MDR...=00003,=00000,=00000)	48
	CALL	OMEGA(R.0003,T.0003,L.0004,=00002,L.0005,=00002)	49
BINARY CARD 00013			
	CALL	I.CMPR(R.0002,T.0002,R.0003,T.0003,R.0002,T.0002)	50
	CALL	I.CAT.(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	51
BINARY CARD 00014			
	CALL	I.SUBS(R.0002,T.0002,MDR...=00003,L.0002,=00002)	52
	CALL	I.SUBS(R.0003,T.0003,MDR...=00003,L.0001,=00002)	53
BINARY CARD 00015			
	CALL	I.NE..(R.0002,T.0002,R.0003,T.0003,R.0002,T.0002)	54
	CALL	I.CAT.(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)	55

Table F.2 cont.

BINARY CARD 00016

CALL I.SUBS(R.0002,T.0002,DR....,=00003,=00000,=00000)
 CALL I.SPEC(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)
 CALL I.SUBS(R.0001,T.0001,MBR....,=00003,=00000,=00000)

56
57
58

BINARY CARD 00017

CALL .OMEGA(R.0002,T.0002,L.0005,=00002,L.0007,=00002)
 CALL I.CMPR(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)

59
60

BINARY CARD 00020

CALL I.SUBS(R.0002,T.0002,MBR....,=00003,=00000,=00000)
 CALL .ALPHA(R.0003,T.0003,L.0005,=00002,L.0007,=00002)

61
62

BINARY CARD 00021

CALL I.CMPR(R.0002,T.0002,R.0003,T.0003,R.0002,T.0002)
 CALL I.SUBS(R.0003,T.0003,MAR....,=00003,L.0008,=00002)

63
64

BINARY CARD 00022

CALL I.MASK(R.0001,T.0001,R.0003,T.0003,R.0002,T.0002,R.0001,
 ETC T.0001)
 CALL I.SUBS(R.0002,T.0002,MDR....,=00003,=00000,=00000)

65
66

BINARY CARD 00023

CALL I.SPEC(R.0001,T.0001,R.0002,T.0002,R.0001,T.0001)
 LITERALS

67
68

00541 000000000000 10000
 00542 000000000002 10000
 00543 000000000003 10000

BINARY CARD 00024

00544 000000000004 10000
 00000 01111

END

72
73

\$DKEND IVCOD.

